1.5

Internals of GNU CC

Richard M. Stallman

Table of Contents

GNU CC GENERAL PUBLIC LICENSE 1
COPYING POLICIES 1
NO WARRANTY 3

1. GNU CC Switches 5

2. Installing GNU CC 9

3. GNU Extensions to the C Language 11

4. GNU CC and Portability 13

5. Interfacing to GNU CC Output 15

6. Passes and Files of the Compiler 17

7. RTL Representation 21
7.1 RTL Object Types 21
7.2 Access to Operands 22
7.3 Machine Modes 24
7.4 Constant Expression Types 25
7.5 Registers and Memory 26
7.6 RTL Expressions for Arithmetic 28
7.7 Comparison Operations 31
7.8 Bit-fields 32
7.9 Conversions 32
7.10 Declarations 34
7.11 Side Effect Expressions 34
7.12 Embedded Side-Effects on Addresses 37
7.13 Insns 38
7.14 Structure Sharing Assumptions 41

8. Machine Descriptions 43
8.1 Instruction Patterns 43
8.2 Example of '`define_insn`' 46
8.3 Operand Constraints 47
8.3.1 Simple Constraints 47
8.3.2 Multiple Alternative Constraints 50
8.3.3 Register Class Preferences 51
8.3.4 Constraint Modifier Characters 51
8.3.5 Not Using Constraints 52
8.4 Standard Insn Names 52
8.5 Interdependence of Patterns 55

9. Machine Description Macros 59
9.1 Run-time Target Specification 59
9.2 Storage Layout 60
9.3 Register Usage 61
9.4 Register Classes 62
9.5 Describing Stack Layout 65
9.6 Addressing Modes 68
9.7 Miscellaneous Parameters 68
9.8 Condition Code Information 70
9.9 Output of Assembler Code 71

Copyright © 1987 Richard M. Stallman.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU CC General Public License" is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled "GNU CC General Public License" may be included in a translation approved by the author instead of in the original English.

GNU CC GENERAL PUBLIC LICENSE

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share GNU CC. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of GNU CC, that you receive source code or else can get it if you want it, that you can change GNU CC or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of GNU CC, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for GNU CC. If GNU CC is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Foundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change GNU CC.

COPYING POLICIES

1. You may copy and distribute verbatim copies of GNU CC source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright © 1987 Free Software Foundation, Inc." (or with the year updated if that is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the GNU CC program a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.
2. You may modify your copy or copies of GNU CC or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

- cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
- cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GNU CC or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).
- You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may copy and distribute GNU CC or any portion of it in compiled, executable or object code form under the terms of Paragraphs 1 and 2 above provided that you do the following:
   - cause each such copy to be accompanied by the corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - cause each such copy to be accompanied by a written offer, with no time limit, to give any third party free (except for a nominal shipping charge) a machine readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - in the case of a recipient of GNU CC in compiled, executable or object code form (without the corresponding source code) you shall cause copies you distribute to be accompanied by a copy of the written offer of source code which you received along with the copy you received.
4. You may not copy, sublicense, distribute or transfer GNU CC except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GNU CC is void and your rights to use the program under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.
5. If you wish to incorporate parts of GNU CC into other free programs whose distribution conditions are different, write to the Free Software Foundation at 1000 Mass Ave, Cambridge, MA 02138. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 1000 Mass Ave, Cambridge, MA 02138, or call (617) 876-3296.

NO WARRANTY

BECAUSE GNU CC IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GNU CC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF GNU CC IS WITH YOU. SHOULD GNU CC PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GNU CC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) GNU CC, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

1. GNU CC Switches

GNU CC compiler uses a command syntax much like the Unix C compiler. The `gcc` program accepts switches and file names as operands. Multiple single-letter switches may *not* be grouped: '`-dr`' is very different from '`-d -r`'.

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. File names which end in '`.c`' are taken as C source to be preprocessed and compiled; compiler output files plus any input files with names ending in '`.s`' are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable.

Command switches allow you to stop this process at an intermediate stage. For example, the '`-c`' switch says not to run the linker. Then the output consists of object files output by the assembler.

Other command switches are passed on to one stage. Some switches control the preprocessor and others the compiler itself. Yet other switches control the assembler and linker; these are not documented here because the GNU assembler and linker are not yet released.

Here are the switches to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

'`-o` *file*' Place linker output in file *file*. The default is '`a.out`' in the current directory.

'`-c`' Compile or assemble the source files, but do not link. Produce object files with names made by replacing '`.c`' or '`.s`' with '`.o`' at the end of the input file names. Do nothing at all for object files specified as input.

'`-S`' Compile into assembler code but do not assemble. The assembler output file name is made by replacing '`.c`' with '`.s`' at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

'`-E`' Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

'`-v`' Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and linker.

'`-B`*prefix*' Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are '`cpp`', '`cc1`', '`as`' and '`ld`'.

For each subprogram to be run, the compiler driver first tries the '`-B`' prefix, if any. If that name is not found, or if '`-B`' was not specified, the driver tries two standard prefixes, which are '`/usr/lib/gcc-`' and '`/usr/local/lib/gcc-`'. If neither of those

results in a file name that is found, the unmodified program name is searched for using the directories specified in your 'PATH' environment variable.

These switches control the details of C compilation itself.

'-O' Optimize. Optimizing compilation takes somewhat more time and a lot more memory.

Without '-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without '-O', only variables declared `register` are allocated in registers. The resulting compiled code is a little worse than produced by PCC without '-O'.

With '-O', the compiler tries to reduce code size and execution time.

A few of the '-f' switches described below turn specific kinds of optimization on or off.

'-g' Produce debugging information in DBX format.

Unlike most other C compilers, GNU CC allows you to use '-g' with '-O'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

'-m*machinespec*'

Machine-dependent switch specifying something about the type of target machine. These switches are defined by the macro `TARGET_SWITCHES` in the machine description. The default for the switches is also defined by that macro, which enables you to change the defaults.

These are the '-m' switches defined in the 68000 machine description:

'-m68020' Generate output for a 68020 (rather than a 68000). This is the default if you use the unmodified sources.

'-m68000' Generate output for a 68000 (rather than a 68020).

'-m68881' Generate output containing 68881 instructions for floating point. This is the default if you use the unmodified sources.

'-msoft-float'

Generate output containing library calls for floating point.

'-mnobitfield'

Do not use the bit-field instructions. '-m68000' implies '-mnobitfield'.

'-mbitfield'

Do use the bit-field instructions. '-m68020' implies '-mbitfield'. This is the default if you use the unmodified sources.

These '-m' switches are defined in the Vax machine description:

'-munix' Do not output certain jump instructions that the Unix assembler for the Vax cannot handle across long ranges.

'-mgnu' Do output those jump instructions, on the assumption that you will assemble with the GNU assembler (which is not yet released).

'-f*flag*' Specify machine-independent flags. These are the flags:

'-ffloat-store'

Do not store floating-point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '-ffloat-store' for such programs.

'-fno-defer-pop'

Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

'-fcombine-regs'

Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to '-O'. I am interested in hearing about the difference this makes.

'-fforce-mem'

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

'-fforce-addr'

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '-fforce-mem' may. I am interested in hearing about the difference this makes.

'-fomit-frame-pointer'

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it

also makes an extra register available in many functions. **It also makes debugging impossible.**

On some machines, such as the Vax, this flag has no effect. This is because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist.

'-d*letters*' Says to make debugging dumps at times specified by *letters*. Here are the possible letters:

'`t`' Dump syntax-tree.

'`r`' Dump after RTL generation.

'`j`' Dump after first jump optimization.

'`s`' Dump after CSE.

'`L`' Dump after loop optimization.

'`f`' Dump after flow analysis.

'`c`' Dump after instruction combination.

'`l`' Dump after local register allocation.

'`g`' Dump after global register allocation.

'`-pedantic`'
Attempt to support strict ANSI standard C. Valid ANSI standard C programs should compile properly with or without this switch. However, without this switch, certain useful or traditional constructs banned by the standard are supported. With this switch, they are rejected. There is no reason to use this switch; it exists only to satisfy pedants.

'`-w`' Inhibit warning messages.

These switches control the C preprocessor, which is run on each C source file before actual compilation.

'`-C`' Tell the preprocessor not to discard comments. Used with the '`-E`' switch.

'`-I`*dir*' Search directory *dir* for include files.

'`-D`*macro*' Define macro *macro* with the empty string as its definition.

'`-D`*macro*`=`*defn*'
Define macro *macro* as *defn*.

'`-U`*macro*' Undefine macro *macro*.

2. Installing GNU CC

1. Choose configuration files.
   - Make a symbolic link from file '`config.h`' to the top-level config file for the machine you are using. Its name should be '`config-`*machine*`.h`'. This file is responsible for defining information about the host machine. It includes '`tm.h`'.
   - Make a symbolic link from '`tm.h`' to the machine-description macro file for your machine (its name should be '`tm-`*machine*`.h`'). For the 68000/68020, do not use '`tm-m68000.h`' directly; instead use one of the files '`tm-sun3.h`', '`tm-sun2.h`', '`tm-isi68.h`' and '`tm-news800.h`'. Each of those files includes '`tm-m68000.h`' but sets up a few things differently as appropriate to the specific model of machine.
   - Make a symbolic link from '`md`' to the machine description pattern file (its name should be '*machine*`.md`').
   - Make a symbolic link from '`aux-output.c`' to the output subroutine file for your machine (its name should be '*machine*`-output.c`').
2. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output file '`parse.tab.c`' is more recent than '`parse.y`' and you do not plan to change '`parse.y`'.)
3. Build the compiler. Just type '`make`' in the compiler directory.
4. Move the first-stage object files into a subdirectory with

   ```
   mkdir stage1
   mv *.o stage1
   ```

   The first-stage executables remain for the next step.

   Once the compiler compiled with itself is working, you may want to delete the subdirectory '`stage1`'.
5. Recompile the compiler with itself, with the command

   ```
   make CC=./gcc CFLAGS="-g -O -I. -B./"
   ```
6. Install the compiler's passes.

   Copy the file '`cc1`' made by the compiler to the name '`/usr/local/lib/gcc-cc1`'.

   Make the file '`/usr/local/lib/gcc-cpp`' either a link to '`/lib/cpp`' or a copy of the file '`cpp`' generated by '`make`'.

   Warning: the GNU CPP may not work for '`ioctl.h`'. This cannot be fixed in the GNU CPP because the bug is in '`ioctl.h`': at least on some machines, it relies on behavior that is incompatible with ANSI C. This behavior consists of substituting for macro argument names when they appear inside of character constants.
7. Install the compiler driver. This is the file '`gcc`' generated by '`make`'.
8. Note that the file '`gcclib.o`' contains subroutines needed at run time by code compiled with GNU CC. Whether they are needed depends on the machine you are using and the code you compile. If you get an undefined-symbol error message from `ld`, include '`gcclib.o`' in the link.

3. GNU Extensions to the C Language

GNU CC provides several language features not found in ANSI standard C. (The '`-pedantic`' switch directs GNU CC to print a warning message if any of these features is used.)

- Compound statements in parentheses may appear inside an expression. This allows you to declare variables within an expression. For example,

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

  is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

  This is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

  But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

- Zero-length arrays are allowed. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline
    = (struct line *) malloc (sizeof (struct line) + this_length);
  thisline->length = thislength;
}
```

  In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

- Variable-size automatic arrays are allowed. These arrays are declared like any other automatic arrays, but with a size that is not a constant expression. The size is computed on entry to the brace-level where the array is declared. The storage is allocated at that time and deallocated when the brace-level is exited. For example,

```
FILE *concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

You can get the same effect with `alloca`, but this is cleaner. Also, space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends.

4. GNU CC and Portability

The main goal of GNU CC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.

GNU CC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, I have not hesitated to define an ad-hoc parameter to the machine description. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake.

GNU CC does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of autoincrement addressing. In the RTL-generation pass, it is often necessary to have multiple strategies for generating code for a particular kind of syntax tree, strategies that are usable for different combinations of parameters. Often I have not tried to address all possible cases, but only the common ones or only the ones that I have encountered. As a result, a new target may require additional strategies. You will know if this happens because the compiler will call `abort`. Fortunately, the new strategies can be added to all versions of the compiler, and will affect only the target machines that need them.

5. Interfacing to GNU CC Output

GNU CC is normally configured to use the same function calling convention normally in use on the target system. This is done with the machine-description macros described (see chapter 9 [Machine Macros], page 59).

However, returning of structure and union values is done differently. As a result, functions compiled with PCC returning such types cannot be called from code compiled with GNU CC, and vice versa. This usually does not cause trouble because the Unix library routines don't return structures and unions.

Structures and unions that are 1, 2, 4 or 8 bytes long are returned in the same registers used for `int` or `double` return values. (GNU CC typically allocates variables of such types in registers also.) Structures and unions of other sizes are returned by storing them into an address passed by the caller in a register. This method is faster than the one normally used by PCC and is also reentrant. The register used for passing the address is specified by the machine-description macro `STRUCTURE_POINTER_REGNUM`.

GNU CC always passes arguments on the stack. At some point it will be extended to pass arguments in registers, for machines which use that as the standard calling convention. This will make it possible to use such a convention on other machines as well. However, that would render it completely incompatible with PCC. We will probably do this once we have a complete GNU system so we can compile the libraries with GNU CC.

If you use `longjmp`, beware of automatic variables. ANSI C says that automatic variables that are not declared `volatile` have undefined values after a `longjmp`. And this is all GNU CC promises to do! It is very difficult to restore register variables correctly, and one of GNU CC's features is that it can put variables in registers without your asking it to.

If you want a variable to be unaltered by `longjmp`, and you don't want to write `volatile` because old C compilers don't accept it, just take the address of the variable. If a variable's address is ever taken, even if just to compute it and ignore it, then the variable cannot go in a register:

```
{
  int careful;
  &careful;
  ...
}
```

Code compiled with GNU CC may call certain library routines. The routines needed on the Vax and 68000 are in the file '`gcclib.c`'. You must compile this file with the standard C compiler, not with GNU CC, and then link it with each program you compile with GNU CC. (In actuality, many programs will not need it.) The usual function call interface is used for calling the library routines. Some standard parts of the C library, such as `bcopy`, are also called automatically.

6. Passes and Files of the Compiler

The overall control structure of the compiler is in 'toplev.c'. This file is responsible for initialization, decoding arguments, opening and closing files, and sequencing the passes.

The parsing pass is invoked only once, to parse the entire input. Each time a the parsing pass reads a complete function definition or top-level data definition, it calls the function rest_of_compilation in 'toplev.c', which is responsible for all further processing necessary, ending with output of the assembler language. All other compiler passes run, in sequence, within rest_of_compilation. After rest_of_compilation returns from compiling a function definition, the storage used for its compilation is entirely freed.

Here is a list of all the passes of the compiler and their source files. Also included is a description of where debugging dumps can be requested with '-d' switches.

- Parsing. This pass reads the entire text of a function definition, constructing a syntax tree. The tree representation does not entirely follow C syntax, because it is intended to support other languages as well.

  C data type analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes.

  Constant folding and associative-law simplifications are also done during this pass.

  The source files of the parsing pass are 'parse.y', 'decl.c', 'typecheck.c', 'stor-layout.c', 'fold-const.c', and 'tree.c'. The last three are intended to be language-independent. There are also header files 'parse.h', 'c-tree.h', 'tree.h' and 'tree.def'. The last two define the format of the tree representation.

  The switch '-dt' causes a debugging dump of the syntax tree after this pass. This dump file's name is made by appending '.tree' to the input file name.

- RTL generation. This pass converts the tree structure for one function into RTL code.

  This is where the bulk of target-parameter-dependent code is found, since often it is necessary for strategies to apply only when certain standard kinds of instructions are available. The purpose of named instruction patterns is to provide this information to the RTL generation pass.

  Optimization is done in this pass for if-conditions that are comparisons, boolean operations or conditional expressions. Tail recursion is detected at this time also. Decisions are made about how best to arrange loops and how to output switch statements.

  The files of the RTL generation pass are 'stmt.c', 'expr.c', 'explow.c', 'expmed.c', 'optabs.c' and 'emit-rtl.c'. Also, the file 'insn-emit.c', generated from the machine description by

the program `genemit`, is used in this pass. The header files '`expr.h`' is used for communication within this pass.

The header files '`insn-flags.h`' and '`insn-codes.h`', generated from the machine description by the programs `genflags` and `gencodes`, tell this pass which standard names are available for use and which patterns correspond to them.

Aside from debugging information output, none of the following passes refers to the tree structure representation of the function.

The switch '`-dr`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.rtl`' to the input file name.

- Jump optimization. This pass simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. It deletes unreferenced labels and unreachable code, except that unreachable code that contains a loop is not recognized as unreachable in this pass. (Such loops are deleted later in the basic block analysis.)

  Jump optimization is performed two or three times. The first time is immediately following RTL generation. The second time is after CSE, but only if CSE says it repeated jump optimization is needed. The last time is right before the final pass. That time, cross-jumping and deletion of no-op move instructions are done together with the optimizations described above.

  The source file of this pass is '`jump.c`'.

  The switch '`-dj`' causes a debugging dump of the RTL code after this pass is run for the first time. This dump file's name is made by appending '`.jump`' to the input file name.

- Register scan. This pass finds the first and last use of each register, as a guide for common subexpression elimination. Its source is in '`regclass.c`'.

- Common subexpression elimination. This pass also does constant propagation. Its source file is '`cse.c`'. If constant propagation causes conditional jumps to become unconditional or to become no-ops, jump optimization is run again when cse is finished.

  The switch '`-ds`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.cse`' to the input file name.

- Loop optimization. This pass moves constant expressions out of loops. Its source file is '`loop.c`'.

  The switch '`-dL`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.loop`' to the input file name.

- Stupid register allocation is performed at this point in a nonoptimizing compilation. It does a little data flow analysis as well. When stupid register allocation is in use, the next pass executed is the reloading pass; the others in between are skipped. The source file is '`stupid.c`'.

- Data flow analysis ('`flow.c`'). This pass divides the program into basic blocks (and in the process deletes unreachable loops); then it computes which pseudo-registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value.

This pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing.

The switch '`-df`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.flow`' to the input file name. If stupid register allocation is in use, this dump file reflects the full results of such allocation.

- Instruction combination ('`combine.c`'). This pass attempts to combine groups of two or three instructions that are related by data flow into single instructions. It combines the RTL expressions for the instructions by substitution, simplifies the result using algebra, and then attempts to match the result against the machine description.

  The switch '`-dc`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.combine`' to the input file name.

- Register class preferencing. The RTL code is scanned to find out which register class is best for each pseudo register. The source file is '`regclass.c`'.

- Local register allocation ('`local-alloc.c`'). This pass allocates hard registers to pseudo registers that are used only within one basic block. Because the basic block is linear, it can use fast and powerful techniques to do a very good job.

  The switch '`-dl`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.lreg`' to the input file name.

- Global register allocation ('`global-alloc.c`'). This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block).

- Reloading. This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register, or has ended up in a register of the wrong kind. It fixes up these instructions by reloading the problematical values temporarily into registers. Additional instructions are generated to do the copying.

  Source files are '`reload.c`' and '`reload1.c`', plus the header '`reload.h`' used for communication between them.

  The switch '`-dg`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.greg`' to the input file name.

- Jump optimization is repeated, this time including cross-jumping and deletion of no-op move instructions.

- Final. This pass outputs the assembler code for the function. It is also responsible for identifying spurious test and compare instructions. The function entry and exit sequences are generated directly as assembler code in this pass; they never exist as RTL.

  The source files are '`final.c`' plus '`insn-output.c`'; the latter is generated automatically from the machine description by the tool '`genoutput`'. The header file '`conditions.h`' is used for communication between these files.

- Debugging information output. This is run after final because it must output the stack slot offsets for pseudo registers that did not get hard registers. Source files are 'dbxout.c' for DBX symbol table format and 'symout.c' for GDB's own symbol table format.

Some additional files are used by all or many passes:

- Every pass uses 'machmode.def', which defines the machine modes.
- All the passes that work with RTL use the header files 'rtl.h' and 'rtl.def', and subroutines in file 'rtl.c'. The tools gen* also use these files to read and work with the machine description RTL.
- Several passes refer to the header file 'insn-config.h' which contains a few parameters (C macro definitions) generated automatically from the machine description RTL by the tool genconfig.
- Several passes use the instruction recognizer, which consists of 'recog.c' and 'recog.h', plus the files 'insn-recog.c' and 'insn-extract.c' that are generated automatically from the machine description by the tools 'genrecog' and 'genextract'.
- Several passes use the header files 'regs.h' which defines the information recorded about pseudo register usage, and 'basic-block.h' which defines the information recorded about basic blocks.
- 'hard-reg-set.h' defines the type HARD_REG_SET, a bit-vector with a bit for each hard register, and some macros to manipulate it. This type is just int if the machine has few enough hard registers; otherwise it is an array of int and some of the macros expand into loops.

7. RTL Representation

Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

7.1 RTL Object Types

RTL uses four kinds of objects: expressions, integers, strings and vectors. Expressions are the most important ones. An RTL expression ("RTX", for short) is a C structure, but it is usually referred to with a pointer; a type that is given the typedef name `rtx`.

An integer is simply an `int`, and a string is a `char *`. Within RTL code, strings appear only inside '`symbol_ref`' expressions, but they appear in other contexts in the RTL expressions that make up machine descriptions. Their written form uses decimal digits.

A string is a sequence of characters. In core it is represented as a `char *` in usual C fashion, and they are written in C syntax as well. However, strings in RTL may never be null. If you write an empty string in a machine description, it is represented in core as a null pointer rather than as a pointer to a null character. In certain contexts, these null pointers instead of strings are valid.

A vector contains an arbitrary, specified number of pointers to expressions. The number of elements in the vector is explicitly present in the vector. The written form of a vector consists of square brackets ('`[...]`') surrounding the elements, in sequence and with whitespace separating them. Vectors of length zero are not created; null pointers are used instead.

Expressions are classified by *expression codes* (also called RTX codes). The expression code is a name defined in '`rtl.def`', which is also (in upper case) a C enumeration constant. The possible expression codes and their meanings are machine-independent. The code of an RTX can be extracted with the macro `GET_CODE (`*x*`)` and altered with `PUT_CODE (`*x*`,` *newcode*`)`.

The expression code determines how many operands the expression contains, and what kinds of objects they are. In RTL, unlike Lisp, you cannot tell by looking at an operand what kind of

object it is. Instead, you must know from its context—from the expression code of the containing expression. For example, in an expression of code 'subreg', the first operand is to be regarded as an expression and the second operand as an integer. In an expression of code 'plus', there are two operands, both of which are to be regarded as expressions. In a 'symbol_ref' expression, there is one operand, which is to be regarded as a string.

Expressions are written as parentheses containing the name of the expression type, its flags and machine mode if any, and then the operands of the expression (separated by spaces).

Expression code names in the 'md' file are written in lower case, but when they appear in C code they are written in upper case. In this manual, they are shown as follows: 'const_int'.

In a few contexts a null pointer is valid where an expression is normally wanted. The written form of this is '(nil)'.

7.2 Access to Operands

For each expression type 'rtl.def' specifies the number of contained objects and their kinds, with four possibilities: 'e' for expression (actually a pointer to an expression), 'i' for integer, 's' for string, and 'E' for vector of expressions. The sequence of letters for an expression code is called its *format*. Thus, the format of 'subreg' is 'ei'.

Two other format characters are used occasionally: 'u' and '0'. 'u' is equivalent to 'e' except that it is printed differently in debugging dumps, and '0' means a slot whose contents do not fit any normal category. '0' slots are not printed at all in dumps, and are often used in special ways by small parts of the compiler.

There are macros to get the number of operands and the format of an expression code:

`GET_RTX_LENGTH (`*code*`)`
: Number of operands of an RTX of code *code*.

`GET_RTX_FORMAT (`*code*`)`
: The format of an RTX of code *code*, as a C string.

Operands of expressions are accessed using the macros XEXP, XINT and XSTR. Each of these macros takes two arguments: an expression-pointer (RTX) and an operand number (counting from zero). Thus,

```
XEXP (x, 2)
```

accesses operand 2 of expression *x*, as an expression.

```
XINT (x, 2)
```

accesses the same operand as an integer. `XSTR`, used in the same fashion, would access it as a string.

Any operand can be accessed as an integer, as an expression or as a string. You must choose the correct method of access for the kind of value actually stored in the operand. You would do this based on the expression code of the containing expression. That is also how you would know how many operands there are.

For example, if *x* is a '`subreg`' expression, you know that it has two operands which can be correctly accessed as `XEXP (x, 0)` and `XINT (x, 1)`. If you did `XINT (x, 0)`, you would get the address of the expression operand but cast as an integer; that might occasionally be useful, but it would be cleaner to write `(int) XEXP (x, 0)`. `XEXP (x, 1)` would also compile without error, and would return the second, integer operand cast as an expression pointer, which would probably result in a crash when accessed. Nothing stops you from writing `XEXP (x, 28)` either, but this will access memory past the end of the expression with unpredictable results.

Access to operands which are vectors is more complicated. You can use the macro `XVEC` to get the vector-pointer itself, or the macros `XVECEXP` and `XVECLEN` to access the elements and length of a vector.

`XVEC (`*exp*`, `*idx*`)`
: Access the vector-pointer which is operand number *idx* in *exp*.

`XVECLEN (`*exp*`, `*idx*`)`
: Access the length (number of elements) in the vector which is in operand number *idx* in *exp*. This value is an `int`.

`XVECLEN (`*exp*`, `*idx*`, `*eltnum*`)`
: Access element number *eltnum* in the vector which is in operand number *idx* in *exp*. This value is an RTX.

  It is up to you to make sure that *eltnum* is not negative and is less than `XVECLEN (`*exp*`, `*idx*`)`.

All the macros defined in this section expand into lvalues and therefore can be used to assign the operands, lengths and vector elements as well as to access them.

7.3 Machine Modes

A machine mode describes a size of data object and the representation used for it. In the C code, machine modes are represented by an enumeration type, `enum machine_mode`, defined in '`machmode.def`'. Each RTL expression has room for a machine mode and so do certain kinds of tree expressions (declarations and types, to be precise).

In debugging dumps and machine descriptions, the machine mode of an RTL expression is written after the expression code with a colon to separate them. The letters '`mode`' which appear at the end of each machine mode name are omitted. For example, `(reg:SI 38)` is a '`reg`' expression with machine mode `SImode`. If the mode is `VOIDmode`, it is not written at all.

Here is a table of machine modes.

`QImode` "Quarter-Integer" mode represents a single byte treated as an integer.

`HImode` "Half-Integer" mode represents a two-byte integer.

`SImode` "Single Integer" mode represents a four-byte integer.

`DImode` "Double Integer" mode represents an eight-byte integer.

`TImode` "Tetra Integer" (?) mode represents a sixteen-byte integer.

`SFmode` "Single Floating" mode represents a single-precision (four byte) floating point number.

`DFmode` "Double Floating" mode represents a double-precision (eight byte) floating point number.

`TFmode` "Tetra Floating" mode represents a quadruple-precision (sixteen byte) floating point number.

`BLKmode` "Block" mode represents values that are aggregates to which none of the other modes apply. In RTL, only memory references can have this mode, and only if they appear in string-move or vector instructions. On machines which have no such instructions, `BLKmode` will not appear in RTL.

`VOIDmode` Void mode means the absence of a mode or an unspecified mode. For example, RTL expressions of code '`const_int`' have mode `VOIDmode` because they can be taken to have whatever mode the context requires. In debugging dumps of RTL, `VOIDmode` is expressed by the absence of any mode.

`EPmode` "Entry Pointer" mode is intended to be used for function variables in Pascal and other block structured languages. Such values contain both a function address and a static chain pointer for access to automatic variables of outer levels. This mode is only partially implemented since C does not use it.

`CSImode`, ... "Complex Single Integer" mode stands for a complex number represented as a pair of

`SImode` integers. Any of the integer and floating modes may have 'C' prefixed to its name to obtain a complex number mode. For example, there are `CQImode`, `CSFmode`, and `CDFmode`. Since C does not support complex numbers, these machine modes are only partially implemented.

`BImode` This is the machine mode of a bit-field in a structure. It is used only in the syntax tree, never in RTL, and in the syntax tree it appears only in declaration nodes. In C, it appears only in `FIELD_DECL` nodes for structure fields defined with a bit size.

The machine description defines `Pmode` as a C macro which expands into the machine mode used for addresses. Normally this is `SImode`.

The only modes which a machine description *must* support are `QImode`, `SImode`, `SFmode` and `DFmode`. The compiler will attempt to use `DImode` for two-word structures and unions, but it would not be hard to program it to avoid this. Likewise, you can arrange for the C type `short int` to avoid using `HImode`. In the long term it would be desirable to make the set of available machine modes machine-dependent and eliminate all assumptions about specific machine modes or their uses from the machine-independent code of the compiler.

Here are some C macros that relate to machine modes:

`GET_MODE (`*x*`)`
Returns the machine mode of the RTX *x*.

`PUT_MODE (`*x*`,` *newmode*`)`
Alters the machine mode of the RTX *x* to be *newmode*.

`GET_MODE_SIZE (`*m*`)`
Returns the size in bytes of a datum of mode *m*.

`GET_MODE_BITSIZE (`*m*`)`
Returns the size in bits of a datum of mode *m*.

`GET_MODE_UNIT_SIZE (`*m*`)`
Returns the size in bits of the subunits of a datum of mode *m*. This is the same as `GET_MODE_SIZE` except in the case of complex modes and `EPmode`. For them, the unit size is the size of the real or imaginary part, or the size of the function pointer or the context pointer.

7.4 Constant Expression Types

The simplest RTL expressions are those that represent constant values.

`(const_int` *i*`)`

This type of expression represents the integer value *i*. *i* is customarily accessed with the macro `INTVAL` as in `INTVAL (`*exp*`)`, which is equivalent to `XINT (`*exp*`, 0)`.

There is only one expression object for the integer value zero; it is the value of the variable `const0_rtx`. Likewise, the only expression for integer value one is found in `const1_rtx`. Any attempt to create an expression of code '`const_int`' and value zero or one will return `const0_rtx` or `const1_rtx` as appropriate.

`(const_double:`*m i0 i1*`)`

Represents a floating point constant value of mode *m*. The two integers *i0* and *i1* together contain the bits of a `double` value. To convert them to a `double`, do

```
union  double d; int i[2]; u;
u.i[0] = XINT (x, 0);
u.i[1] = XINT (x, 1);
```

and then refer to `u.d`. The value of the constant is represented as a double in this fashion even if the value represented is single-precision.

The global variables `dconst0_rtx` and `fconst0_rtx` hold '`const_double`' expressions with value 0, in modes `DFmode` and `SFmode`, respectively.

`(symbol_ref` *symbol*`)`

Represents the value of an assembler label for data. *symbol* is a string that describes the name of the assembler label. If it starts with a '`*`', the label is the rest of *symbol* not including the '`*`'. Otherwise, the label is *symbol*, prefixed with '`_`'.

`(label_ref` *label*`)`

Represents the value of an assembler label for code. It contains one operand, an expression, which must be a '`code_label`' that appears in the instruction sequence to identify the place where the label should go.

The reason for using a distinct expression type for code label references is so that jump optimization can distinguish them.

`(const` *exp*`)`

Represents a constant that is the result of an assembly-time arithmetic computation. The operand, *exp*, is an expression that contains only constants ('`const_int`', '`symbol_ref`' and '`label_ref`' expressions) combined with '`plus`' and '`minus`'. However, not all combinations are valid, since the assembler cannot do arbitrary arithmetic on relocatable symbols.

7.5 Registers and Memory

Here are the RTL expression types for describing access to machine registers and to main memory.

`(reg:`*m n*`)`

For small values of the integer *n* (less than `FIRST_PSEUDO_REGISTER`), this stands for a reference to machine register number *n*: a *hard register*. For larger values of *n*, it stands for a temporary value or *pseudo register*. The compiler's strategy is to generate code assuming an unlimited number of such pseudo registers, and later convert them into hard registers or into memory references.

The symbol `FIRST_PSEUDO_REGISTER` is defined by the machine description, since the number of hard registers on the machine is an invariant characteristic of the machine. Note, however, that not all of the machine registers must be general registers. All the machine registers that can be used for storage of data are given hard register numbers, even those that can be used only in certain instructions or can hold only certain types of data.

Each pseudo register number used in a function's RTL code is represented by a unique '`reg`' expression.

m is the machine mode of the reference. It is necessary because machines can generally refer to each register in more than one mode. For example, a register may contain a full word but there may be instructions to refer to it as a half word or as a single byte, as well as instructions to refer to it as a floating point number of various precisions.

Even for a register that the machine can access in only one mode, the mode must always be specified.

A hard register may be accessed in various modes throughout one function, but each pseudo register is given a natural mode and is accessed only in that mode. When it is necessary to describe an access to a pseudo register using a nonnatural mode, a '`subreg`' expression is used.

A '`reg`' expression with a machine mode that specifies more than one word of data may actually stand for several consecutive registers. If in addition the register number specifies a hardware register, then it actually represents several consecutive hardware registers starting with the specified one.

Such multi-word hardware register '`reg`' expressions may not be live across the boundary of a basic block. The lifetime analysis pass does not know how to record properly that several consecutive registers are actually live there, and therefore register allocation would be confused. The CSE pass must go out of its way to make sure the situation does not arise.

`(subreg:`*m reg wordnum*`)`

'`subreg`' expressions are used to refer to a register in a machine mode other than its natural one, or to refer to one register of a multi-word '`reg`' that actually refers to several registers.

Each pseudo-register has a natural mode. If it is necessary to operate on it in a different mode—for example, to perform a fullword move instruction on a pseudo-register that

contains a single byte— the pseudo-register must be enclosed in a 'subreg'. In such a case, *wordnum* is zero.

The other use of 'subreg' is to extract the individual registers of a multi-register value. Machine modes such as DImode and EPmode indicate values longer than a word, values which usually require two consecutive registers. To access one of the registers, use a 'subreg' with mode SImode and a *wordnum* that says which register.

The compilation parameter WORDS_BIG_ENDIAN, if defined, says that word number zero is the most significant part; otherwise, it is the least significant part.

Note that it is not valid to access a DFmode value in SFmode using a 'subreg'. On some machines the most significant part of a DFmode value does not have the same format as a single-precision floating value.

(cc0) This refers to the machine's condition code register. It has no operands and may not have a machine mode. It may be validly used in only two contexts: as the destination of an assignment (in test and compare instructions) and in comparison operators comparing against zero ('const_int' with value zero; that is to say, const0_rtx).

There is only one expression object of code 'cc0'; it is the value of the variable cc0_rtx. Any attempt to create an expression of code 'cc0' will return cc0_rtx.

One special thing about the condition code register is that instructions can set it implicitly. On many machines, nearly all instructions set the condition code based on the value that they compute or store. It is not necessary to record these actions explicitly in the RTL because the machine description includes a prescription for recognizing the instructions that do so (by means of the macro NOTICE_UPDATE_CC). Only instructions whose sole purpose is to set the condition code, and instructions that use the condition code, need mention (cc0).

(pc) This represents the machine's program counter. It has no operands and may not have a machine mode. (pc) may be validly used only in certain specific contexts in jump instructions.

There is only one expression object of code 'pc'; it is the value of the variable pc_rtx. Any attempt to create an expression of code 'pc' will return pc_rtx.

All instructions that do not jump alter the program counter implicitly by incrementing it, but there is no need to mention this in the RTL.

(mem:*m addr*)

This RTX represents a reference to main memory at an address represented by the expression *addr*. *m* specifies how large a unit of memory is accessed.

7.6 RTL Expressions for Arithmetic

`(plus:`*m* *x* *y*`)`

Represents the sum of the values represented by *x* and *y* carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*.

`(minus:`*m* *x* *y*`)`

Like '`plus`' but represents subtraction.

`(minus` *x* *y*`)`

Represents the result of subtracting *y* from *x* for purposes of comparison. The absence of a machine mode in the '`minus`' expression indicates that the result is computed without overflow, as if with infinite precision.

Of course, machines can't really subtract with infinite precision. However, they can pretend to do so when only the sign of the result will be used, which is the case when the result is stored in (`cc0`). And that is the only way this kind of expression may validly be used: as a value to be stored in the condition codes.

`(neg:`*m* *x*`)`

Represents the negation (subtraction from zero) of the value represented by *x*, carried out in mode *m*. *x* must be valid for mode *m*.

`(mult:`*m* *x* *y*`)`

Represents the signed product of the values represented by *x* and *y* carried out in machine mode *m*. If *x* and *y* are both valid for mode *m*, this is ordinary size-preserving multiplication. Alternatively, both *x* and *y* may be valid for a different, narrower mode. This represents the kind of multiplication that generates a product wider than the operands. Widening multiplication and same-size multiplication are completely distinct and supported by different machine instructions; machines may support one but not the other.

'`mult`' may be used for floating point division as well. Then *m* is a floating point machine mode.

`(umult:`*m* *x* *y*`)`

Like '`mult`' but represents unsigned multiplication. It may be used in both same-size and widening forms, like '`mult`'. '`umult`' is used only for fixed-point multiplication.

`(div:`*m* *x* *y*`)`

Represents the quotient in signed division of *x* by *y*, carried out in machine mode *m*. If *m* is a floating-point mode, it represents the exact quotient; otherwise, the integerized quotient. If *x* and *y* are both valid for mode *m*, this is ordinary size-preserving division. Some machines have division instructions in which the operands and quotient widths are not all the same; such instructions are represented by '`div`' expressions in which the machine modes are not all the same.

`(udiv:`*m* *x* *y*`)`

Like '`div`' but represents unsigned division.

`(mod:`*m x y*`)`
`(umod:`*m x y*`)`
Like '`div`' and '`udiv`' but represent the remainder instead of the quotient.

`(not:`*m x*`)`
Represents the bitwise complement of the value represented by *x*, carried out in mode *m*, which must be a fixed-point machine mode. *x* must be valid for mode *m*, which must be a fixed-point mode.

`(and:`*m x y*`)`
Represents the bitwise logical-and of the values represented by *x* and *y*, carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*, which must be a fixed-point mode.

`(ior:`*m x y*`)`
Represents the bitwise inclusive-or of the values represented by *x* and *y*, carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*, which must be a fixed-point mode.

`(xor:`*m x y*`)`
Represents the bitwise exclusive-or of the values represented by *x* and *y*, carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*, which must be a fixed-point mode.

`(lshift:`*m x c*`)`
Represents the result of logically shifting *x* left by *c* places. *x* must be valid for the mode *m*, a fixed-point machine mode. *c* must be valid for a fixed-point mode; which mode is determined by the mode called for in the machine description entry for the left-shift instruction. For example, on the Vax, the mode of *c* is `QImode` regardless of *m*.

On some machines, negative values of *c* may be meaningful; this is why logical left shift an arithmetic left shift are distinguished. For example, Vaxes have no right-shift instructions, and right shifts are represented as left-shift instructions whose counts happen to be negative constants or else computed (in a previous instruction) by negation.

`(ashift:`*m x c*`)`
Like '`lshift`' but for arithmetic left shift.

`(lshiftrt:`*m x c*`)`
`(ashiftrt:`*m x c*`)`
Like '`lshift`' and '`ashift`' but for right shift.

`(rotate:`*m x c*`)`
`(rotatert:`*m x c*`)`
Similar but represent left and right rotate.

`(abs:`*m x*`)`
Represents the absolute value of *x*, computed in mode *m*. *x* must be valid for *m*.

`(sqrt:`*m* *x*`)`

Represents the square root of *x*, computed in mode *m*. *x* must be valid for *m*. Most often *m* will be a floating point mode.

7.7 Comparison Operations

Comparison operators test a relation on two operands and are considered to represent the value 1 if the relation holds, or zero if it does not. The mode of the comparison is determined by the operands; they must both be valid for a common machine mode. A comparison with both operands constant would be invalid as the machine mode could not be deduced from it, but such a comparison should never exist in RTL due to constant folding.

Inequality comparisons come in two flavors, signed and unsigned. Thus, there are distinct expression codes '`gt`' and '`gtu`' for signed and unsigned greater-than. These can produce different results for the same pair of integer values: for example, 1 is signed greater-than -1 but not unsigned greater-than, because -1 when regarded as unsigned is actually `0xffffffff` which is greater than 1.

The signed comparisons are also used for floating point values. Floating point comparisons are distinguished by the machine modes of the operands.

The comparison operators may be used to compare the condition codes (`cc0`) against zero, as in `(eq (cc0) (const_int 0))`. Such a construct actually refers to the result of the preceding instruction in which the condition codes were set. The above example stands for 1 if the condition codes were set to say "zero" or "equal", 0 otherwise. Although the same comparison operators are used for this as may be used in other contexts on actual data, no confusion can result since the machine description would never allow both kinds of uses in the same context.

`(eq` *x* *y*`)` 1 if the values represented by *x* and *y* are equal, otherwise 0.

`(ne` *x* *y*`)` 1 if the values represented by *x* and *y* are not equal, otherwise 0.

`(gt` *x* *y*`)` 1 if the *x* is greater than *y*. If they are fixed-point, the comparison is done in a signed sense.

`(gtu` *x* *y*`)`

Like '`gt`' but does unsigned comparison, on fixed-point numbers only.

`(lt` *x* *y*`)`

`(ltu` *x* *y*`)`

Like '`gt`' and '`gtu`' but test for "less than".

(ge *x y*)

(geu *x y*)

Like 'gt' and 'gtu' but test for "greater than or equal".

(le *x y*)

(leu *x y*)

Like 'gt' and 'gtu' but test for "less than or equal".

(if_then_else *cond then else*)

This is not a comparison operation but is listed here because it is always used in conjunction with a comparison operation. To be precise, *cond* is a comparison expression. This expression represents a choice, according to *cond*, between the value represented by *then* and the one represented by *else*.

On most machines, 'if_then_else' expressions are valid only to express conditional jumps.

7.8 Bit-fields

Special expression codes exist to represent bit-field instructions. These types of expressions are lvalues in RTL; they may appear on the left side of a assignment, indicating insertion of a value into the specified bit field.

(sign_extract:SI *loc size pos*)

This represents a reference to a sign-extended bit-field contained or starting in *loc* (a memory or register reference). The bit field is *size* bits wide and starts at bit *pos*. The compilation switch BITS_BIG_ENDIAN says which end of the memory unit *pos* counts from.

Which machine modes are valid for *loc* depends on the machine, but typically *loc* should be a single byte when in memory or a full word in a register.

(zero_extract:SI *loc pos size*)

Like 'sign_extract' but refers to an unsigned or zero-extended bit field. The same sequence of bits are extracted, but they are filled to an entire word with zeros instead of by sign-extension.

7.9 Conversions

All conversions between machine modes must be represented by explicit conversion operations. For example, an expression which the sum of a byte and a full word cannot be written as (plus:SI

`(reg:QI 34) (reg:SI 80))` because the 'plus' operation requires two operands of the same machine mode. Therefore, the byte-sized operand is enclosed in a conversion operation, as in

```
(plus:SI (sign_extend:SI (reg:QI 34)) (reg:SI 80))
```

The conversion operation is not a mere placeholder, because there may be more than one way of converting from a given starting mode to the desired final mode. The conversion operation code says how to do it.

`(sign_extend:`*m* *x*`)`
: Represents the result of sign-extending the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode narrower than *m*.

`(zero_extend:`*m* *x*`)`
: Represents the result of zero-extending the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode narrower than *m*.

`(float_extend:`*m* *x*`)`
: Represents the result of extending the value *x* to machine mode *m*. *m* must be a floating point mode and *x* a floating point value of a mode narrower than *m*.

`(truncate:`*m* *x*`)`
: Represents the result of truncating the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode wider than *m*.

`(float_truncate:`*m* *x*`)`
: Represents the result of truncating the value *x* to machine mode *m*. *m* must be a floating point mode and *x* a floating point value of a mode wider than *m*.

`(float:`*m* *x*`)`
: Represents the result of converting fixed point value *x*, regarded as signed, to floating point mode *m*.

`(unsigned_float:`*m* *x*`)`
: Represents the result of converting fixed point value *x*, regarded as unsigned, to floating point mode *m*.

`(fix:`*m* *x*`)`
: When *m* is a fixed point mode, represents the result of converting floating point value *x* to mode *m*, regarded as signed. How rounding is done is not specified, so this operation may be used validly in compiling C code only for integer-valued operands.

`(unsigned_fix:`*m* *x*`)`
: Represents the result of converting floating point value *x* to fixed point mode *m*, regarded as signed. How rounding is done is not specified.

`(fix:`*m* *x*`)`
: When *m* is a floating point mode, represents the result of converting floating point

value x (valid for mode m) to an integer, still represented in floating point mode m, by rounding towards zero.

7.10 Declarations

Declaration expression codes do not represent arithmetic operations but rather state assertions about their operands.

`(volatile:`*m* *x*`)`

Represents the same value x does, but makes the assertion that it should be treated as a volatile value. This forbids coalescing multiple accesses or deleting them even if it would appear to have no effect on the program. x must be a '`mem`' expression with mode m.

The first thing the reload pass does to an insn is to remove all '`volatile`' expressions from it; each one is replaced by its operand.

Recognizers will never recognize anything with '`volatile`' in it. This automatically prevents some optimizations on such things (such as instruction combination). After the reload pass removes all volatility information, the insns can be recognized.

Cse removes '`volatile`' from destinations of '`set`'s, because no optimizations reorder such '`set`'s. This is not required for correct code and is done to permit some optimization on the value to be stored.

`(unchanging:`*m* *x*`)`

Represents the same value x does, but makes the assertion that its value is effectively constant during the execution of the current function. This permits references to x to be moved freely within the function. x must be a '`reg`' expression with mode m.

`(strict_low_part (subreg:`*m* `(reg:`*n* *r*`) 0))`

This expression code is used in only one context: operand 0 of a '`set`' expression. In addition, the operand of this expression must be a '`subreg`' expression.

The presence of '`strict_low_part`' says that the part of the register which is meaningful in mode n but is not part of mode m is not to be altered. Normally, an assignment to such a subreg is allowed to have undefined effects on the rest of the register when m is less than a word.

7.11 Side Effect Expressions

The expression codes described so far represent values, not actions. But machine instructions

never produce values; they are meaningful only for their side effects on the state of the machine. Special expression codes are used to represent side effects.

The body of an instruction is always one of these side effect codes; the codes described above, which represent values, appear only as the operands of these.

`(set` *lval x*`)`

Represents the action of storing the value of *x* into the place represented by *lval*. *lval* must be an expression representing a place that can be stored in: '`reg`' (or '`subreg`' or '`strict_low_part`'), '`mem`', '`pc`' or '`cc0`'.

If *lval* is a '`reg`', '`subreg`' or '`mem`', it has a machine mode; then *x* must be valid for that mode.

If *lval* is a '`reg`' whose machine mode is less than the full width of the register, then it means that the part of the register specified by the machine mode is given the specified value and the rest of the register receives an undefined value. Likewise, if *lval* is a '`subreg`' whose machine mode is narrower than `SImode`, the rest of the register can be changed in an undefined way.

If *lval* is a '`strict_low_part`' of a '`subreg`', then the part of the register specified by the machine mode of the '`subreg`' is given the value *x* and the rest of the register is not changed.

If *lval* is `(cc0)`, it has no machine mode, and *x* may have any mode. This represents a "test" or "compare" instruction.

If *lval* is `(pc)`, we have a jump instruction, and the possibilities for *x* are very limited. It may be a '`label_ref`' expression (unconditional jump). It may be an '`if_then_else`' (conditional jump), in which case either the second or the third operand must be `(pc)` (for the case which does not jump) and the other of the two must be a '`label_ref`' (for the case which does jump). *x* may also be a '`mem`' or `(plus:SI (pc) y)`, where *y* may be a '`reg`' or a '`mem`'; these unusual patterns are used to represent jumps through branch tables.

`(return)` Represents a return from the current function, on machines where this can be done with one instruction, such as Vaxes. On machines where a multi-instruction "epilogue" must be executed in order to return from the function, returning is done by jumping to a label which precedes the epilogue, and the '`return`' expression code is never used.

`(call` *function nargs*`)`

Represents a function call. *function* is a '`mem`' expression whose address is the address of the function to be called. *nargs* is an expression representing the number of words of argument.

Each machine has a standard machine mode which *function* must have. The machine description defines macro `FUNCTION_MODE` to expand into the requisite mode name.

The purpose of this mode is to specify what kind of addressing is allowed, on machines where the allowed kinds of addressing depend on the machine mode being addressed.

`(clobber x)`

Represents the storing or possible storing of an unpredictable, undescribed value into *x*, which must be a '`reg`' or '`mem`' expression.

One place this is used is in string instructions that store standard values into particular hard registers. It may not be worth the trouble to describe the values that are stored, but it is essential to inform the compiler that the registers will be altered, lest it attempt to keep data in them across the string instruction.

x may also be null—a null C pointer, no expression at all. Such a `(clobber (null))` expression means that all memory locations must be presumed clobbered.

Note that the machine description classifies certain hard registers as "call-clobbered". All function call instructions are assumed by default to clobber these registers, so there is no need to use '`clobber`' expressions to indicate this fact. Also, each function call is assumed to have the potential to alter any memory location.

`(use x)` Represents the use of the value of *x*. It indicates that the value in *x* at this point in the program is needed, even though it may not be apparent why this is so. Therefore, the compiler will not attempt to delete instructions whose only effect is to store a value in *x*. *x* must be a '`reg`' expression.

`(parallel [x0 x1 ...])`
Represents several side effects performed in parallel. The square brackets stand for a vector; the operand of '`parallel`' is a vector of expressions. *x0*, *x1* and so on are individual side effects—expressions of code '`set`', '`call`', '`return`', '`clobber`' or '`use`'.

"In parallel" means that first all the values used in the individual side-effects are computed, and second all the actual side-effects are performed. For example,

```
(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1)))
           (set (mem:SI (reg:SI 1)) (reg:SI 1))])
```

says unambiguously that the values of hard register 1 and the memory location addressed by it are interchanged. In both places where `(reg:SI 1)` appears as a memory address it refers to the value in register 1 *before* the execution of the instruction.

Three expression codes appear in place of a side effect, as the body of an insn, though strictly speaking they do not describe side effects as such:

`(asm_input s)`

Represents literal assembler code as described by the string *s*.

`(addr_vec:m [lr0 lr1 ...])`
Represents a table of jump addresses. *lr0* etc. are '`label_ref`' expressions. The mode *m* specifies how much space is given to each address; normally *m* would be `Pmode`.

`(addr_diff_vec:`*m base* `[`*lr0 lr1* `...])`
Represents a table of jump addresses expressed as offsets from *base*. *lr0* etc. are '`label_ref`' expressions and so is *base*. The mode *m* specifies how much space is given to each address-difference.

7.12 Embedded Side-Effects on Addresses

Four special side-effect expression codes appear as memory addresses.

`(pre_dec:`*m x*`)`
Represents the side effect of decrementing *x* by a standard amount and represents also the value that *x* has after being decremented. *x* must be a '`reg`' or '`mem`', but most machines allow only a '`reg`'. *m* must be the machine mode for pointers on the machine in use. The amount *x* is decrement by is the length in bytes of the machine mode of the containing memory reference of which this expression serves as the address. Here is an example of its use:

```
(mem:DF (pre_dec:SI (reg:SI 39)))
```

This says to decrement pseudo register 39 by the length of a `DFmode` value and use the result to address a `DFmode` value.

`(pre_inc:`*m x*`)`
Similar, but specifies incrementing *x* instead of decrementing it.

`(post_dec:`*m x*`)`
Represents the same side effect as '`pre_decrement`' but a different value. The value represented here is the value *x* has *before* being decremented.

`(post_inc:`*m x*`)`
Similar, but specifies incrementing *x* instead of decrementing it.

These embedded side effect expressions must be used with care. Instruction patterns may not use them. Until the '`flow`' pass of the compiler, they may occur only to represent pushes onto the stack. The '`flow`' pass finds cases where registers are incremented or decremented in one instruction and used as an address shortly before or after; these cases are then transformed to use pre- or post-increment or -decrement.

Explicit popping of the stack could be represented with these embedded side effect operators, but that would not be safe; the instruction combination pass could move the popping past pushes, thus changing the meaning of the code.

An instruction that can be represented with an embedded side effect could also be represented using 'parallel' containing an additional 'set' to describe how the address register is altered. This is not done because machines that allow these operations at all typically allow them wherever a memory address is called for. Describing them as additional parallel stores would require doubling the number of entries in the machine description.

7.13 Insns

The RTL representation of the code for a function is a doubly-linked chain of objects called *insns*. Insns are expressions with special codes that are used for no other purpose. Some insns are actual instructions; others represent dispatch tables for `switch` statements; others represent labels to jump to or various sorts of declarative information.

In addition to its own specific data, each insn must have a unique id-number that distinguishes it from all other insns in the current function, and chain pointers to the preceding and following insns. These three fields occupy the same position in every insn, independent of the expression code of the insn. They could be accessed with `XEXP` and `XINT`, but instead three special macros are always used:

`INSN_UID` (*i*)
: Accesses the unique id of insn *i*.

`PREV_INSN` (*i*)
: Accesses the chain pointer to the insn preceding *i*. If *i* is the first insn, this is a null pointer.

`NEXT_INSN` (*i*)
: Accesses the chain pointer to the insn following *i*. If *i* is the last insn, this is a null pointer.

The `NEXT_INSN` and `PREV_INSN` pointers must always correspond: if *i* is not the first insn,

```
NEXT_INSN (PREV_INSN (insn)) == insn
```

is always true.

Every insn has one of the following six expression codes:

'`insn`'
: The expression code '`insn`' is used for instructions that do not jump and do not do

function calls. Insns with code 'insn' have four additional fields beyond the three mandatory ones listed above. These four are described in a table below.

'jump_insn'
: The expression code 'jump_insn' is used for instructions that may jump (or, more generally, may contain 'label_ref' expressions). 'jump_insn' insns have the same extra fields as 'insn' insns, accessed in the same way.

'call_insn'
: The expression code 'call_insn' is used for instructions that may do function calls. It is important to distinguish these instructions because they imply that certain registers and memory locations may be altered unpredictably.

  'call_insn' insns have the same extra fields as 'insn' insns, accessed in the same way.

'code_label'
: A 'code_label' insn represents a label that a jump insn can jump to. It contains one special field of data in addition to the three standard ones. It is used to hold the *label number*, a number that identifies this label uniquely among all the labels in the compilation (not just in the current function). Ultimately, the label is represented in the assembler output as an assembler label 'L*n*' where *n* is the label number.

'barrier'
: Barriers are placed in the instruction stream after unconditional jump instructions to indicate that the jumps are unconditional. They contain no information beyond the three standard fields.

'note'
: 'note' insns are used to represent additional debugging and declarative information. They contain two nonstandard fields, an integer which is accessed with the macro NOTE_LINE_NUMBER and a string accessed with NOTE_SOURCE_FILE.

  If NOTE_LINE_NUMBER is positive, the note represents the position of a source line and NOTE_SOURCE_FILE is the source file name that the line came from. These notes control generation of line number data in the assembler output.

  Otherwise, NOTE_LINE_NUMBER is not really a line number but a code with one of the following values (and NOTE_SOURCE_FILE must contain a null pointer):

  NOTE_INSN_DELETED
  : Such a note is completely ignorable. Some passes of the compiler delete insns by altering them into notes of this kind.

  NOTE_INSN_BLOCK_BEG
  NOTE_INSN_BLOCK_END
  : These types of notes indicate the position of the beginning and end of a level of scoping of variable names. They control the output of debugging information.

  NOTE_INSN_LOOP_BEG
  NOTE_INSN_LOOP_END
  : These types of notes indicate the position of the beginning and end of a

`while` or `for` loop. They enable the loop optimizer to find loops quickly.

Here is a table of the extra fields of '`insn`', '`jump_insn`' and '`call_insn`' insns:

`PATTERN` (*i*)
: An expression for the side effect performed by this insn.

`REG_NOTES` (*i*)
: A list (chain of '`expr_list`' expressions) giving information about the usage of registers in this insn. This list is set up by the flow analysis pass; it is a null pointer until then.

`LOG_LINKS` (*i*)
: A list (chain of '`insn_list`' expressions) of previous "related" insns: insns which store into registers values that are used for the first time in this insn. (An additional constraint is that neither a jump nor a label may come between the related insns). This list is set up by the flow analysis pass; it is a null pointer until then.

`INSN_CODE` (*i*)
: An integer that says which pattern in the machine description matches this insn, or -1 if the matching has not yet been attempted.

  Such matching is never attempted and this field is not used on an insn whose pattern consists of a single '`use`', '`clobber`', '`asm`', '`addr_vec`' or '`addr_diff_vec`' expression.

The `LOG_LINKS` field of an insn is a chain of '`insn_list`' expressions. Each of these has two operands: the first is an insn, and the second is another '`insn_list`' expression (the next one in the chain). The last '`insn_list`' in the chain has a null pointer as second operand. The significant thing about the chain is which insns appear in it (as first operands of '`insn_list`' expressions). Their order is not significant.

The `REG_NOTES` field of an insn is a similar chain but of '`expr_list`' expressions instead of '`insn_list`'. The first operand is a '`reg`' RTX. Its presence in the list can have three possible meanings, distinguished by a value that is stored in the machine-mode field of the '`expr_list`' because that is a conveniently available space, but that is not really a machine mode. These values belong to the C type `enum reg_note` and there are three of them:

`REG_DEAD` The '`reg`' listed dies in this insn; that is to say, altering the value immediately after this insn would not affect the future behavior of the program.

`REG_INC` The '`reg`' listed is incremented (or decremented; at this level there is no distinction) by an embedded side effect inside this insn.

`REG_CONST`
: The '`reg`' listed has a value that could safely be replaced everywhere by the value that this insn copies into it. ("Safely" here refers to the data flow of the program; such

replacement may require reloading into registers for some of the insns in which the '`reg`' is replaced.)

`REG_WAS_O`

The '`reg`' listed contained zero before this insn. You can rely on this note if it is present; its absence implies nothing.

(The only difference between the expression codes '`insn_list`' and '`expr_list`' is that the first operand of an '`insn_list`' is assumed to be an insn and is printed in debugging dumps as the insn's unique id; the first operand of an '`expr_list`' is printed in the ordinary way as an expression.)

7.14 Structure Sharing Assumptions

The compiler assumes that certain kinds of RTL expressions are unique; there do not exist two distinct objects representing the same value. In other cases, it makes an opposite assumption: that no RTL expression object of a certain kind appears in more than one place in the containing structure.

These assumptions refer to a single function; except for the RTL objects that describe global variables and external functions, no RTL objects are common to two functions.

- Each pseudo-register has only a single '`reg`' object to represent it, and therefore only a single machine mode.
- For any symbolic label, there is only one '`symbol_ref`' object referring to it.
- There is only one '`const_int`' expression with value zero, and only one with value one.
- There is only one '`pc`' expression.
- There is only one '`cc0`' expression.
- There is only one '`const_double`' expression with mode `SFmode` and value zero, and only one with mode `DFmode` and value zero.
- No '`label_ref`' appears in more than one place in the RTL structure; in other words, it is safe to do a tree-walk of all the insns in the function and assume that each time a '`label_ref`' is seen it is distinct from all other '`label_ref`'s seen.
- Aside from the cases listed above, the only kind of expression object that may appear in more than one place is the '`mem`' object that describes a stack slot or a static variable.

8. Machine Descriptions

A machine description has two parts: a file of instruction patterns ('`.md`' file) and a C header file of macro definitions.

The '`.md`' file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). It may also contain comments. A semicolon causes the rest of the line to be a comment, unless the semicolon is inside a quoted string.

See the next chapter for information on the C header file.

8.1 Instruction Patterns

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a '`define_insn`' expression.

Sometimes an insn can match more than one instruction pattern. Then the pattern that appears first in the machine description is the one used. Therefore, more specific patterns (patterns that will match fewer things) and faster instructions (those that will produce better code when they do match) should usually go first in the description.

In some cases the effect of ordering the patterns can be used to hide a pattern when it is not valid. For example, the 68000 has an instruction for converting a fullword to floating point and another for converting a byte to floating point. An instruction converting an integer to floating point could match either one. We put the pattern to convert the fullword first to make sure that one will be used rather than the other. (Otherwise a large integer might be generated as a single-byte immediate quantity, which would not work.) Instead of using this pattern ordering it would be possible to make the pattern for convert-a-byte smart enough to deal properly with any constant value.

The '`define_insn`' expression contains four operands:

1. An optional name. The presence of a name indicate that this instruction pattern can perform a certain standard job for the RTL-generation pass of the compiler. This pass knows certain

names and will use the instruction patterns with those names, if the names are defined in the machine description.

The absence of a name is indicated by writing an empty string where the name should go. Nameless instruction patterns are never used for generating RTL code, but they may permit several simpler insns to be combined later on.

Names that are not thus known and used in RTL-generation have no effect; they are equivalent to no name at all.

2. The recognition template. This is a vector of incomplete RTL expressions which show what the instruction should look like. It is incomplete because it may contain '`match_operand`' and '`match_dup`' expressions that stand for operands of the instruction.

   If the vector has only one element, that element is what the instruction should look like. If the vector has multiple elements, then the instruction looks like a '`parallel`' expression containing that many elements as described.

3. A condition. This is a string which contains a C expression that is the final test to decide whether an insn body matches this pattern.

   For a named pattern, the condition (if present) may not depend on the data in the insn being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.

   For nameless patterns, the condition is applied only when matching an individual insn, and only after the insn has matched the pattern's recognition template. The insn's operands may be found in the vector `operands`.

4. A string that says how to output matching insns as assembler code. In the simpler case, the string is an output template, much like a `printf` control string. '%' in the string specifies where to insert the operands of the instruction; the '%' is followed by an operand number, or by a letter and then an operand number.

   '%c*digit*' can be used to substitute an operand that is a constant value without the syntax that normally indicates an immediate operand.

   '%n*digit*' is like '%c*digit*' except that the value of the constant is negated before printing.

   '%a*digit*' can be used to substitute an operand as if it were a memory reference, with the actual operand treated as the address. This may be useful when outputting a "load address" instruction, because often the assembler syntax for such an instruction requires you to write the operand as if it were a memory reference.

   '%l*digit*' is used to substitute a `label_ref` into a jump instruction.

   '%%' outputs a '%' into the assembler code.

   You may define your own machine-specific letters for use in '%' constructs. Any nonstandard letter (or punctuation character) that follows a '%' will be passed to the `PRINT_OPERAND` macro that you have defined, along with the number that follows. The macro can give the letter any interpretation you desire.

If you omit the operand number, zero is assumed. You should not omit the operand number unless the operand is truly irrelevant. One example is in the machine-dependent specification '%.' used for the 68000, which means '.' if using Motorola assembler syntax, no output for MIT syntax. Because no operand value is relevant to '%.', it is appropriate to omit the operand number.

The template may generate multiple assembler instructions. Write the text for the instructions, with '\;' between them.

If the output control string starts with a '*', then it is not an output template but rather a piece of C program that should compute a template. It should execute a `return` statement to return the template-string you want. Most such templates use C string literals, which require doublequote characters to delimit them. To include these doublequote characters in the string, prefix each one with '\'.

The operands may be found in the array `operands`, whose C data type is `rtx []`.

It is possible to output an assembler instruction and then go on to output or compute more of them, using the subroutine `output_asm_insn`. This receives two arguments: a template-string and a vector of operands. The vector may be `operands`, or it may be another array of `rtx` that you declare locally and initialize yourself.

The recognition template is used also, for named patterns, for constructing insns. Construction involves substituting specified operands into a copy of the template. Matching involves determining the values that serve as the operands in the insn being matched. Both of these activities are controlled by two special expression types that direct matching and substitution of the operands.

`(match_operand:`*m n testfn constraint*`)`

This expression is a placeholder for operand number *n* of the insn. When constructing an insn, operand number *n* will be substituted at this point. When matching an insn, whatever appears at this position in the insn will be taken as operand number *n*; but it must satisfy *testfn* or this instruction pattern will not match at all.

Operand numbers must be chosen consecutively counting from zero in each instruction pattern. There may be only one '`match_operand`' expression in the pattern for each expression number, and they must appear in order of increasing expression number.

testfn is a string that is the name of a C function that accepts two arguments, a machine mode and an expression. During matching, the function will be called with *m* as the mode argument and the putative operand as the other argument. If it returns zero, this instruction pattern fails to match. *testfn* may be an empty string; then it means no test is to be done on the operand.

Most often, *testfn* is "`general_operand`". It checks that the putative operand is either a constant, a register or a memory reference, and that it is valid for mode *m*.

constraint is explained later.

`(match_dup n)`

This expression is also a placeholder for operand number *n*. It is used when the operand needs to appear more than once in the insn.

In construction, '`match_dup`' behaves exactly like '`match_operand`': the operand is substituted into the insn being constructed. But in matching, '`match_dup`' behaves differently. It assumes that operand number *n* has already been determined by a '`match_operand`' appearing earlier in the recognition template, and it matches only an identical-looking expression.

`(address (match_operand:`*m* *n* `"address_operand" ""))`

This complex of expressions is a placeholder for an operand number *n* in a "load address" instruction: an operand which specifies a memory location in the usual way, but for which the actual operand value used is the address of the location, not the contents of the location.

'`address`' expressions never appear in RTL code, only in machine descriptions. And they are used only in machine descriptions that do not use the operand constraint feature. When operand constraints are in use, the letter 'p' in the constraint serves this purpose.

m is the machine mode of the *memory location being addressed*, not the machine mode of the address itself. That mode is always the same on a given target machine (it is `Pmode`, which normally is `SImode`), so there is no point in mentioning it; thus, no machine mode is written in the '`address`' expression. If some day support is added for machines in which addresses of different kinds of objects appear differently or are used differently (such as the PDP-10), different formats would perhaps need different machine modes and these modes might be written in the '`address`' expression.

8.2 Example of '`define_insn`'

Here is an actual example of an instruction pattern, for the 68000/68020.

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  "*
{ if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
  return \"cmpl #0,%0\"; }")
```

This is an instruction that sets the condition codes based on the value of a general operand. It

has no condition, so any insn whose RTL description has the form shown may be handled according to this pattern. The name '`tstsi`' means "test a `SImode` value" and tells the RTL generation pass that, when it is necessary to test such a value, an insn to do so can be constructed using this pattern.

The output control string is a piece of C code which chooses which output template to return based on the kind of operand and the specific type of CPU for which code is being generated.

'`"rm"`' is an operand constraint. Its meaning is explained below.

8.3 Operand Constraints

Each '`match_operand`' in an instruction pattern can specify a constraint for the type of operands allowed. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

8.3.1 Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

'`m`' A memory operand is allowed, with any kind of address that the machine supports in general.

'`o`' A memory operand is allowed, but only if the address is *offsetable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address. For example, an address which is constant is offsetable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsetable. More complicated indirect/indexed addresses may or may not be offsetable depending on the other addressing modes that the machine supports.

'`<`' A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.

'>' A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

'r' A register operand is allowed provided that it is in a general register.

'd', 'a', 'f', ... Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. 'd', 'a' and 'f' are defined on the 68000/68020 to stand for data, address and floating point registers.

'i' An immediate integer operand (one with constant value) is allowed.

'I', 'J', 'K', ... Other letters in the range 'I' through 'M' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, 'I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

'F' An immediate floating operand (expression code 'const_double') is allowed.

'G', 'H' 'G' and 'H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

's' An immediate integer operand whose value is not an explicit integer is allowed. This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use 's' instead of 'i'? Sometimes it allows better code to be generated. For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -32 and 31, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a 'moveq' instruction. We arrange for this to happen by defining the letter 'K' to mean "any integer outside the range -32 to 31", and then specifying 'Ks' in the operand constraints.

'g' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

'*n*' (a digit)
An operand identical to operand number *n* is allowed. If a digit is used together with letters, the digit should come last.

'p' An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

If 'p' is used in the constraint, the test-function in the 'match_operand' must be address_operand.

In order to have valid assembler code, each operand must satisfy its constraint. But a failure to do so does not prevent the pattern from applying to an insn. Instead, it directs the compiler to

modify the code so that the constraint will be satisfied. Usually this is done by copying an operand into a register.

Contrast, therefore, the two instruction patterns that follow:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "r")
        (plus:SI (match_dup 0)
                 (match_operand:SI 1 "general_operand" "r")))]
  ""
  "...")
```

which has two operands, one of which must appear in two places, and

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "r")
        (plus:SI (match_operand:SI 1 "general_operand" "0")
                 (match_operand:SI 2 "general_operand" "r")))]
  ""
  "...")
```

which has three operands, two of which are required by a constraint to be identical. If we are considering an insn of the form

```
(insn n prev next
  (set (reg:SI 3)
       (plus:SI (reg:SI 6) (reg:SI 109)))
  ...)
```

the first pattern would not apply at all, because this insn does not contain two identical subexpressions in the right place. The pattern would say, "That does not look like an add instruction; try other patterns." The second pattern would say, "Yes, that's an add instruction, but there is something wrong with it." It would direct the reload pass of the compiler to generate additional insns to make the constraint true. The results might look like this:

```
(insn n2 prev n
  (set (reg:SI 3) (reg:SI 6))
  ...)

(insn n n2 next
  (set (reg:SI 3)
       (plus:SI (reg:SI 3) (reg:SI 109)))
  ...)
```

Because insns that don't fit the constraints are fixed up by loading operands into registers, every instruction pattern's constraints must permit the case where all the operands are in registers. It need not permit all classes of registers; the compiler knows how to copy registers into other registers of the proper class in order to make an instruction valid. But if no registers are permitted, the compiler will be stymied: it does not know how to save a register in memory in order to make an instruction valid. Instruction patterns that reject registers can be made valid by attaching a condition-expression that refuses to match an insn at all if the crucial operand is a register.

8.3.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative. Here is how it is done for fullword logical-or on the 68000:

```
(define_insn "iorsi3"
  [(set (match_operand:SI 0 "general_operand" "=%m,d")
        (ior:SI (match_operand:SI 1 "general_operand" "0,0")
                (match_operand:SI 2 "general_operand" "dKs,dmKs")))]
  ...)
```

The first alternative has '`m`' (memory) for operand 0, '`0`' for operand 1 (meaning it must match operand 0), and '`dKs`' for operand 2. The second alternative has '`d`' (data register) for operand 0, '`0`' for operand 1, and '`dmKs`' for operand 2. The '`=`' and '`%`' in the constraint for operand 0 are not part of any alternative; their meaning is explained in the next section.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the '`?`' and '`!`' characters:

'`?`' Disparage slightly the alternative that the '`?`' appears in, as a choice when no alternative

applies exactly. The compiler regards this alternative as one unit more costly for each '?' that appears in it.

'!' Disparage severely the alternative that the '!' appears in. When operands must be copied into registers, the compiler will never choose this alternative as the one to strive for.

8.3.3 Register Class Preferences

The operand constraints have another function: they enable the compiler to decide which kind of hardware register a pseudo register is best allocated to. The compiler examines the constraints that apply to the insns that use the pseudo register, looking for the machine-dependent letters such as 'd' and 'a' that specify classes of registers. The pseudo register is put in whichever class gets the most "votes". The constraint letters 'g' and 'r' also vote: they vote in favor of a general register. The machine description says which registers are considered general.

Of course, on some machines all registers are equivalent, and no register classes are defined. Then none of this complexity is relevant.

8.3.4 Constraint Modifier Characters

'=' Means that this operand is written by the instruction, but its previous value is not used.

'+' Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. '=' identifies an output; '+' identifies an operand that is both input and output; all other operands are assumed to be input only.

'%' Declares the instruction to be commutative for operands 1 and 2. This means that the compiler may interchange operands 1 and 2 if that will make the operands fit their constraints.

'#' Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

'*' Says that the following character should be ignored when choosing register preferences. '*' has no effect on the meaning of the constraint as a constraint.

8.3.5 Not Using Constraints

Some machines are so clean that operand constraints are not required. For example, on the Vax, an operand valid in one context is valid in any other context. On such a machine, every operand constraint would be 'g', excepting only operands of "load address" instructions which are written as if they referred to a memory location's contents but actual refer to its address. They would have constraint 'p'.

For such machines, instead of writing 'g' and 'p' for all the constraints, you can choose to write a description with empty constraints. Then you write '""' for the constraint in every 'match_operand'. Address operands are identified by writing an 'address' expression around the 'match_operand', not by their constraints.

When the machine description has just empty constraints, certain parts of compilation are skipped, making the compiler faster.

8.4 Standard Insn Names

Here is a table of the instruction names that are meaningful in the RTL generation pass of the compiler. Giving one of these names to an instruction pattern tells the RTL generation pass that it can use the pattern in to accomplish a certain task.

'movm' Here *m* is a two-letter machine mode name, in lower case. This instruction pattern moves data with that machine mode from operand 1 to operand 0. For example, 'movsi' moves full-word data.

If operand 0 is a 'subreg' with mode *m* of a register whose natural mode is wider than *m*, the effect of this instruction is to store the specified value in the part of the register that corresponds to mode *m*. The effect on the rest of the register is undefined.

'movstrictm'

Like 'movm' except that if operand 0 is a 'subreg' with mode *m* of a register whose natural mode is wider, the 'movstrictm' instruction is guaranteed not to alter any of the register except the part which belongs to mode *m*.

'addm3' Add operand 2 and operand 1, storing the result in operand 0. All operands must have mode *m*. This can be used even on two-address machines, by means of constraints requiring operands 1 and 0 to be the same location.

'subm3', 'mulm3', 'umulm3', 'divm3', 'udivm3', 'modm3', 'umodm3', 'andm3', 'iorm3', 'xorm3'

Similar, for other arithmetic operations.

'`andcbm3`' Bitwise logical-and operand 1 with the complement of operand 2 and store the result in operand 0.

'`mulhisi3`'
Multiply operands 1 and 2, which have mode `HImode`, and store a `SImode` product in operand 0.

'`mulqihi3`', '`mulsidi3`'
Similar widening-multiplication instructions of other widths.

'`umulqihi3`', '`umulhisi3`', '`umulsidi3`'
Similar widening-multiplication instructions that do unsigned multiplication.

'`divmodm4`'
Signed division that produces both a quotient and a remainder. Operand 1 is divided by operand 2 to produce a quotient stored in operand 0 and a remainder stored in operand 3.

'`udivmodm4`'
Similar, but does unsigned division.

'`divmodmn4`'
Like '`divmodm4`' except that only the dividend has mode *m*; the divisor, quotient and remainder have mode *n*. For example, the Vax has a '`divmoddisi4`' instruction (but it is omitted from the machine description, because it is so slow that it is faster to compute remainders by the circumlocution that the compiler will use if this instruction is not available).

'`ashlm3`' Arithmetic-shift operand 1 left by a number of bits specified by operand 2, and store the result in operand 0. Operand 2 has mode `SImode`, not mode *m*.

'`ashrm3`', '`lshlm3`', '`lshrm3`', '`rotlm3`', '`rotrm3`'
Other shift and rotate instructions.

'`negm2`' Negate operand 1 and store the result in operand 0.

'`absm2`' Store the absolute value of operand 1 into operand 0.

'`sqrtm2`' Store the square root of operand 1 into operand 0.

'`one_cmplm2`'
Store the bitwise-complement of operand 1 into operand 0.

'`cmpm`' Compare operand 0 and operand 1, and set the condition codes.

'`tstm`' Compare operand 0 against zero, and set the condition codes.

'`movstrm`' Block move instruction. The addresses of the destination and source strings are the first two operands, and both are in mode `Pmode`. The number of bytes to move is the third operand, in mode *m*.

'`cmpstrm`' Block compare instruction, with operands like '`movstrm`' except that the two memory blocks are compared byte by byte in lexicographic order. The effect of the instruction is to set the condition codes.

'`floatmn2`'
: Convert operand 1 (valid for fixed point mode *m*) to floating point mode *n* and store in operand 0 (which has mode *n*).

'`fixmn2`'
: Convert operand 1 (valid for floating point mode *m*) to fixed point mode *n* as a signed number and store in operand 0 (which has mode *n*). This instruction's result is defined only when the value of operand 1 is an integer.

'`fixunsmn2`'
: Convert operand 1 (valid for floating point mode *m*) to fixed point mode *n* as an unsigned number and store in operand 0 (which has mode *n*). This instruction's result is defined only when the value of operand 1 is an integer.

'`ftruncm2`'
: Convert operand 1 (valid for floating point mode *m*) to an integer value, still represented in floating point mode *m*, and store it in operand 0 (valid for floating point mode *m*).

'`fix_truncmn2`'
: Like '`fixmn2`' but works for any floating point value of mode *m* by converting the value to an integer.

'`fixuns_truncmn2`'
: Like '`fixunsmn2`' but works for any floating point value of mode *m* by converting the value to an integer.

'`truncmn`'
: Truncate operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point or both floating point.

'`extendmn`'
: Sign-extend operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point or both floating point.

'`zero_extendmn`'
: Zero-extend operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point.

'`extv`'
: Extract a bit-field from operand 1 (a register or memory operand), where operand 2 specifies the width in bits and operand 3 the starting bit, and store it in operand 0. Operand 0 must have `Simode`. Operand 1 may have mode `QImode` or `SImode`; often `SImode` is allowed only for registers. Operands 2 and 3 must be valid for `SImode`.

  The RTL generation pass generates this instruction only with constants for operands 2 and 3.

  The bit-field value is sign-extended to a full word integer before it is stored in operand 0.

'`extzv`'
: Like '`extv`' except that the bit-field value is zero-extended.

'`insv`'
: Store operand 3 (which must be valid for `SImode`) into a bit-field in operand 0, where operand 1 specifies the width in bits and operand 2 the starting bit. Operand 0 may

have mode `QImode` or `SImode`; often `SImode` is allowed only for registers. Operands 1 and 2 must be valid for `SImode`.

The RTL generation pass generates this instruction only with constants for operands 1 and 2.

'*scond*' Store zero or nonzero in the operand according to the condition codes. Value stored is nonzero iff the condition *cond* is true. *cond* is the name of a comparison operation expression code, such as '`eq`', '`lt`' or '`leu`'.

You specify the mode that the operand must have when you write the `match_operand` expression. The compiler automatically sees which mode you have used and supplies an operand of that mode.

The value stored for a true condition must have 1 as its low bit. Otherwise the instruction is not suitable and must be omitted from the machine description. You must tell the compiler exactly which value is stored by defining the macro `STORE_FLAG_VALUE`.

'b*cond*' Conditional branch instruction. Operand 0 is a '`label_ref`' that refers to the label to jump to. Jump if the condition codes meet condition *cond*.

'`call`' Subroutine call instruction. Operand 1 is the number of arguments and operand 0 is the function to call. Operand 0 should be a '`mem`' RTX whose address is the address of the function.

'`return`' Subroutine return instruction. This instruction pattern name should be defined only if a single instruction can do all the work of returning from a function.

'`tablejump`'

'`casem`'

8.5 Interdependence of Patterns

Every machine description must have a named pattern for each of the conditional branch names 'b*cond*'. The recognition template must always have the form

```
(set (pc)
     (if_then_else (cond (cc0) (const_int 0))
                   (label_ref (match_operand 0 "" ""))
                   (pc)))
```

In addition, every machine description must have an anonymous pattern for each of the possible reverse-conditional branches. These patterns look like

```
(set (pc)
```

```
(if_then_else (cond [(cc0) (const_int 0)])
              (pc)
              (label_ref (match_operand 0 "" "")))
```

They are necessary because jump optimization can turn direct-conditional branches into reverse-conditional branches.

The compiler does more with RTL than just create it from patterns and recognize the patterns: it can perform arithmetic expression codes when constant values for their operands can be determined. As a result, sometimes having one pattern can require other patterns. For example, the Vax has no 'and' instruction, but it has 'and not' instructions. Here is the definition of one of them:

```
(define_insn "andcbsi2"
  [(set (match_operand:SI 0 "general_operand" "")
        (and:SI (match_dup 0)
                (not:SI (match_operand:SI
                         1 "general_operand" ""))))]
  ""
  "bicl2 %1,%0")
```

If operand 1 is an explicit integer constant, an instruction constructed using that pattern can end up looking like

```
(set (reg:SI 41)
     (and:SI (reg:SI 41)
             (const_int 0xffff7fff)))
```

(where the integer constant is the one's complement of what appeared in the original instruction).

To avoid a fatal error, the compiler must have a pattern that recognizes such an instruction. Here is what is used:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "")
        (and:SI (match_dup 0)
                (match_operand:SI 1 "general_operand" "")))]
  "GET_CODE (operands[1]) == CONST_INT"
  "*
 operands[1]
   = gen_rtx (CONST_INT, VOIDmode, ~INTVAL (operands[1]));
  return \"bicl2 %1,%0\";
")
```

Whereas a pattern to match a general 'and' instruction is impossible to support on the Vax, this pattern is possible because it matches only a constant second argument: a special case that can be output as an 'and not' instruction.

In some cases machines support instructions identical except for the machine mode of one or more operands. For example, there may be "sign-extend halfword" and "sign-extend byte" instructions whose patterns are

```
(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:HI 1 ...)))

(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:QI 1 ...)))
```

Constant integers do not specify a machine mode, so an instruction to extend a constant value could match either pattern. The pattern it actually will match is the one that appears first in the file. For correct results, this must be the one for the widest possible mode (`HImode`, here). If the pattern matches the `QImode` instruction, the results will be incorrect if the constant value does not actually fit that mode.

Such instructions to extend constants are rarely generated because they are optimized away, but they do occasionally happen in nonoptimized compilations.

9. Machine Description Macros

The other half of the machine description is a C header file conventionally given the name '`tm-`*machine*`.h`'. The file '`tm.h`' should be a link to it. The header file '`config.h`' includes '`tm.h`' and most compiler source files include '`config.h`'.

9.1 Run-time Target Specification

`CPP_PREDEFINES`

Define this to be a string constant containing '`-D`' switches to define the predefined macros that identify this machine and system.

For example, on the Sun, one can use the value

```
"-Dmc68000 -Dsun"
```

`extern int target_flags;`

This declaration should be present.

`TARGET_...` This series of macros is to allow compiler command arguments to enable or disable the use of optional features of the target machine. For example, one machine description serves both the 68000 and the 68020; a command argument tells the compiler whether it should use 68020-only instructions or not. This command argument works by means of a macro `TARGET_68020` that tests a bit in `target_flags`.

Define a macro `TARGET_`*featurename* for each such option. Its definition should test a bit in `target_flags`; for example:

```
#define TARGET_68020 (target_flags & 1)
```

One place where these macros are used is in the condition-expressions of instruction patterns. Note how `TARGET_68020` appears frequently in the 68000 machine description file, '`m68000.md`'. Another place they are used is in the definitions of the other macros in the '`tm-`*machine*`.h`' file.

`TARGET_SWITCHES`

This macro defines names of command switches to set and clear bits in `target_flags`. Its definition is an initializer with a subgrouping for each command switches.

Each subgrouping contains a string constant, that defines the switch name, and a number, which contains the bits to set in `target_flags`. A negative number says to clear bits instead; the negative of the number is which bits to clear. The actual switch name is made by appending '`-m`' to the specified name.

One of the subgroupings should have a null string. The number in this grouping is the default value for `target_flags`. Any target switches act starting with that value.

Here is an example which defines '-m68000' and '-m68020' with opposite meanings, and picks the latter as the default:

```
#define TARGET_SWITCHES \
  { { "68020", 1},      \
    { "68000", -1},     \
    { "", 1}}
```

9.2 Storage Layout

`BITS_BIG_ENDIAN`

Define this macro if the most significant bit in a byte has the lowest number. This means that bit-field instructions count from the most significant bit. If the machine has no bit-field instructions, this macro is irrelevant.

`BYTES_BIG_ENDIAN`

Define this macro if the most significant byte in a word has the lowest number.

`WORDS_BIG_ENDIAN`

Define this macro if, in a multiword object, the most significant word has the lowest number.

`BITS_PER_UNIT`

Number of bits in an addressable storage unit (byte); normally 8.

`BITS_PER_WORD`

Number of bits in a word; normally 32.

`UNITS_PER_WORD`

Number of storage units in a word; normally 4.

`POINTER_SIZE`

Width of a pointer, in bits.

`PARM_BOUNDARY`

Alignment required for pointers, in bits.

`FUNCTION_BOUNDARY`

Alignment required for a function entry point, in bits.

`BIGGEST_ALIGNMENT`

Biggest alignment that anything can require on this machine, in bits.

`STRICT_ALIGNMENT`

Define this if instructions will fail to work if given data not on the nominal alignment. If instructions will merely go slower in that case, do not define this macro.

9.3 Register Usage

`FIRST_PSEUDO_REGISTER`

Number of hardware registers known to the compiler. They receive numbers 0 through `FIRST_PSEUDO_REGISTER-1`; thus, the first pseudo register's number really is assigned the number `FIRST_PSEUDO_REGISTER`.

`FIXED_REGISTERS`

An initializer that says which registers are used for fixed purposes all throughout the compiled code and are therefore not available for general allocation. These would include the stack pointer, the frame pointer, the program counter on machines where that is considered one of the addressable registers, and any other numbered register with a standard use.

This information is expressed as a sequence of numbers, separated by commas and surrounded by braces. The *n*th number is 1 if register *n* is fixed, 0 otherwise

`CALL_USED_REGISTERS`

Like `FIXED_REGISTERS` but has 1 for each register that is clobbered (in general) by function calls as well as for fixed registers. This macro therefore identifies the registers that are not available for general allocation of values that must live across function calls.

If a registers has 0 in `CALL_USED_REGISTERS`, the compiler automatically saves it on function entry and restores it on function exit, if the register is used within the function.

`HARD_REGNO_REGS (`*regno*`, `*mode*`)`

A C expression for the number of consecutive hard registers, starting at register number *regno*, required to hold a value of mode *mode*.

On a machine where all registers are exactly one word, a suitable definition of this macro is

```
#define HARD_REGNO_NREGS(REGNO, MODE)            \
   ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1)  \
    / UNITS_PER_WORD))
```

`HARD_REGNO_MODE_OK (`*regno*`, `*mode*`)`

A C expression that is nonzero if it is permissible to store a value of mode *mode* in hard register number *regno* (or in several registers starting with that one). For a machine where all registers are equivalent, a suitable definition is

```
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1
```

It is not necessary for this macro to check for fixed register numbers because the allocation mechanism considers them to be always occupied.

`MODES_TIEABLE_P (`*mode1*`, `*mode2*`)`

A C expression that is nonzero if it is desirable to choose register allocation so as to avoid move instructions between a value of mode *mode1* and a value of mode *mode2*.

If `HARD_REGNO_MODE_OK` (*r*, *mode1*) and `HARD_REGNO_MODE_OK` (*r*, *mode2*) are ever different for any *r*, then `MODES_TIEABLE_P` (*mode1*, *mode2*) must be zero.

`PC_REGNUM`

If the program counter has a register number, define this as that register number. Otherwise, do not define it.

`STACK_POINTER_REGNUM`

The register number of the stack pointer register, which must also be a fixed register according to `FIXED_REGISTERS`. On many machines, the hardware determines which register this is.

`FRAME_POINTER_REGNUM`

The register number of the frame pointer register, which is used to access automatic variables in the stack frame. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose.

If the frame pointer register is a fixed register according to `FIXED_REGISTERS`, then the frame pointer is always maintained in every function. If the frame pointer register is not marked as a fixed register, then under certain circumstances, when a function does not need to keep its frame pointer explicitly, this register may be used for other things.

`ARG_POINTER_REGNUM`

The register number of the arg pointer register, which is used to access the function's argument list. On some machines, this is the same as the frame pointer register. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose. It must in any case be a fixed register according to `FIXED_REGISTERS`.

`STATIC_CHAIN_REGNUM`

The register number used for passing a function's static chain pointer. This is needed for languages such as Pascal and Algol where functions defined within other functions can access the local variables of the outer functions; it is not currently used because C does not provide this feature.

The static chain register need not be a fixed register.

`STRUCT_VALUE_REGNUM`

When a function's value's mode is `BLKmode`, the value is not returned according to `FUNCTION_VALUE`. Instead, the caller passes the address of a block of memory in which the value should be stored. `STRUCT_VALUE_REGNUM` is the register in which this address is passed.

9.4 Register Classes

On many machines, the numbered registers are not all equivalent. For example, certain registers

may not be allowed for indexed addressing; certain registers may not be allowed in some instructions. These machine restrictions are described to the compiler using *register classes*.

You define a number of register classes, giving each one a name and saying which of the registers belong to it. Then you can specify register classes that are allowed as operands to particular instruction patterns.

In general, each register will belong to several classes. In fact, one class must be named `ALL_REGS` and contain all the registers. Another class must be named `NO_REGS` and contain no registers. Often the union of two classes will be another class; however, this is not required.

One of the classes must be named `GENERAL_REGS`. There is nothing terribly special about the name, but the operand constraint letters 'r' and 'g' specify this class. If `GENERAL_REGS` is the same as `ALL_REGS`, just define it as a macro which expands to `ALL_REGS`.

The way classes other than `GENERAL_REGS` are specified in operand constraints is through machine-dependent operand constraint letters. You can define such letters to correspond to various classes, then use them in operand constraints.

You must also specify certain redundant information about the register classes: for each class, which classes contain it and which ones are contained in it; for each pair of classes, the largest class contained in their union.

`enum reg_class`

An enumeral type that must be defined with all the register class names as enumeral values. `NO_REGS` must be first. `ALL_REGS` must be the last register class, followed by one more enumeral value, `LIM_REG_CLASSES`, which is not a register class but rather tells how many classes there are.

Each register class has a number, which is the value of casting the class name to type `int`. The number serves as an index in many of the tables described below.

`REG_CLASS_NAMES`

An initializer containing the names of the register classes as C string constants. These names are used in writing some of the debugging dumps.

`REG_CLASS_CONTENTS`

An initializer containing the contents of the register classes, as integers which are bit masks. The *n*th integer specifies the contents of class *n*. The way the integer *mask* is interpreted is that register *r* is in the class if *mask* `&` `(1 <<` *r*`)` is 1.

When the machine has more than 32 registers, an integer does not suffice. Then the integers are replaced by sub-initializers, braced groupings containing several integers.

Each sub-initializer must be suitable as an initializer for the type `HARD_REG_SET` which is defined in '`hard-reg-set.h`'.

`REGNO_REG_CLASS (`*regno*`)`

A C expression whose value is a register class containing hard register *regno*. In general there is more that one such class; choose a class which is *minimal*, meaning that no smaller class also contains the register.

`REG_CLASS_SUPERCLASSES`

A two-level initializer that says, for each class, which classes contain it. The *n*th element of the initializer is a sub-initializer for class *n*; it contains the names of the other classes that contain class *n* (but not the name of class *n* itself), followed by `LIM_REG_CLASSES` to mark the end of the element.

`REG_CLASS_SUBCLASSES`

Similar to `REG_CLASS_SUPERCLASSES`, except that element *n* lists the classes *contained in* class *n*, followed once again by `LIM_REG_CLASSES` to mark the end of the element.

`REG_CLASS_SUBUNION`

An two-level initializer for a two-dimensional array. The element (*m*, *n*) of this array must be a class that is "close to" being the union of classes *m* and *n*. If there is a class that is exactly that union, use it; otherwise, choose some smaller class, preferably as large as possible but certainly not containing any register that is neither in class *m* nor in class *n*.

`INDEX_REG_CLASS`

A macro whose definition is the name of the class to which a valid index register must belong.

`REG_CLASS_FROM_LETTER (`*char*`)`

A C expression which defines the machine-dependent operand constraint letters for register classes. If *char* is such a letter, the value should be the register class corresponding to it. Otherwise, the value should be `NO_REGS`.

`REGNO_OK_FOR_CLASS_P (`*regno*`,` *class*`)`

A C expression which is nonzero if register number *regno* is a hard register belonging to class *class*. The expression is always zero if *regno* is a pseudo register.

`REG_OK_FOR_CLASS_P (`*reg*`,` *class*`)`

A C expression which is nonzero if *reg* (an RTX assumed to have code '`reg`') belongs to class *class*.

What about pseudo registers? There are two alternatives, and the machine description header file must be able to do either one on command. If the macro `REG_OK_STRICT` is defined, this macro should be defined to reject all pseudo registers (return 0 for them). Otherwise, this macro should be defined to accept all pseudo registers (return 1 for them).

Some source files of the compiler define `REG_OK_STRICT` before including the machine

description header file, while others do not, according to the needs of that part of the compiler.

`PREFERRED_RELOAD_CLASS (`*x*`, `*class*`)`

A C expression that places additional restrictions on the register class to use when it is necessary to copy value *x* into a register in class *class*. The value is a register class; perhaps *class*, or perhaps another, smaller class. *class* is always safe as a value. In fact, the definition

```
#define PREFERRED_RELOAD_CLASS(X,CLASS) CLASS
```

is always safe. However, sometimes returning a more restrictive class makes better code. For example, on the 68000, when *x* is an integer constant that is in range for a 'moveq' instruction, the value of this macro is always `DATA_REGS` as long as *class* includes the data registers. Requiring a data register guarantees that a 'moveq' will be used.

Two other special macros

`CONST_OK_FOR_LETTER_P (`*value*`, `*c*`)`

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of integer values. If *c* is one of those letters, the expression should check that *value*, an integer, is in the appropriate range and return 1 if so, 0 otherwise. If *c* is not one of those letters, the value should be 0 regardless of *value*.

`CONST_DOUBLE_OK_FOR_LETTER_P (`*value*`, `*c*`)`

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of floating values. If *c* is one of those letters, the expression should check that *value*, an RTX of code 'const_double', is in the appropriate range and return 1 if so, 0 otherwise. If *c* is not one of those letters, the value should be 0 regardless of *value*.

9.5 Describing Stack Layout

`STACK_GROWS_DOWNWARD`

Define this macro if pushing a word onto the stack moves the stack pointer to a smaller address.

When we say, "define this macro if ...," it means that the compiler checks this macro only with `#ifdef` so the precise definition used does not matter.

`FRAME_GROWS_DOWNWARD`

Define this macro if the addresses of local variable slots are at negative offsets from the frame pointer.

`STARTING_FRAME_OFFSET`

Offset from the frame pointer to the first local variable slot to be allocated.

If `FRAME_GROWS_DOWNWARD`, the next slot's offset is found by subtracting the length of the first slot from `STARTING_FRAME_OFFSET`. Otherwise, it is found by adding the length of the first slot to the value `STARTING_FRAME_OFFSET`.

`PUSH_ROUNDING (`*npushed*`)`

A C expression that is the number of bytes actually pushed onto the stack when an instruction attempts to push *npushed* bytes.

On some machines, the definition

```
#define PUSH_ROUNDING(BYTES) (BYTES)
```

will suffice. But on other machines, instructions that appear to push one byte actually push two bytes in an attempt to maintain alignment. Then the definition should be

```
#define PUSH_ROUNDING(BYTES) (((BYTES) + 1) & ~1)
```

`FIRST_PARM_OFFSET`

Offset from the argument pointer register to the first argument's address.

`RETURN_POPS_ARGS`

Define this macro if returning from a function automatically pops the function's arguments. Do not define it if the caller must pop them.

`FUNCTION_VALUE (`*valtype*`,` *func*`)`

A C expression to create an RTX representing the place where a function returns a value of data type *valtype*. *valtype* is a tree node representing a data type. Write `TYPE_MODE (`*valtype*`)` to get the machine mode used to represent that type. On many machines, only the mode is relevant. (Actually, on most machines, scalar values are returned in the same place regardless of mode).

If the precise function being called is known, *func* is a tree node (`FUNCTION_DECL`) for it; otherwise, *func* is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

`LIBCALL_VALUE (`*mode*`)`

A C expression to create an RTX representing the place where a standard C-compiler library function returns a value of mode *mode*. If the precise function being called is known, *func* is a tree node (`FUNCTION_DECL`) for it; otherwise, *func* is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

`FUNCTION_VALUE_REGNO_P (`*regno*`)`

A C expression that is nonzero if *regno* is the number of a hard register in which function values are sometimes returned.

A register whose use for returning values is limited to when it is the second of a pair (for a value of type `double`, say) need not be recognized by this macro. So for most machines, this definition suffices:

```
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 0)
```

`FUNCTION_PROLOGUE (`*file*`, `*size*`)`

A C compound statement that outputs the assembler code for entry to a function. The prologue is responsible for setting up the stack frame, initializing the frame pointer register, saving registers that must be saved, and allocating *size* additional bytes of storage for the local variables. *size* is an integer. *file* is a stdio stream to which the assembler code should be output.

The label for the beginning of the function need not be output by this macro. That has already been done when the macro is run.

To determine which registers to save, the macro can refer to the array `regs_ever_live`: element *r* is nonzero if hard register *r* is used anywhere within the function. This implies the function prologue should save register *r*, but not if it is one of the call-used registers.

`EXIT_IGNORES_STACK`

Define this macro if the return instruction or the function epilogue ignores the value of the stack pointer; in other words, if it is safe to delete an instruction to adjust the stack pointer before a return from the function.

Note that this macro applies to functions for which frame pointers are maintained. It is never possible to delete a stack adjustment in a function that has no frame pointer, and the compiler knows that regardless of whether `EXIT_IGNROES_STACK` is defined.

`FUNCTION_EPILOGUE (`*file*`, `*size*`)`

A C compound statement that outputs the assembler code for exit from a function. The epilogue is responsible for restoring the saved registers and stack pointer to their values when the function was called, and returning control to the caller. This macro takes the same arguments as the macro `FUNCTION_PROLOGUE`, and the registers to restore are determined from `regs_ever_live` and `CALL_USED_REGISTERS` in the same way.

On some machines, there is a single instruction that does all the work of returning from the function. On these machines, give that instruction the name '`return`' and do not define the macro `FUNCTION_EPILOGUE` at all.

`FIX_FRAME_POINTER_ADDRESS (`*addr*`)`

A C compound statement to alter a memory address that uses the frame pointer register so that it uses the stack pointer register instead. This must be done in the instructions that load parameter values into registers, when the reload pass determines that a frame pointer is not necessary for the function. *addr* will be a C variable name, and the updated address should be stored in that variable.

Even if your machine description specifies there will always be a frame pointer in the frame pointer register, you must still define `FIX_FRAME_POINTER_ADDRESS`, but the definition will never be executed and may be empty.

9.6 Addressing Modes

`HAVE_POST_INCREMENT`

Define this macro if the machine supports post-increment addressing.

`HAVE_PRE_INCREMENT`
`HAVE_POST_DECREMENT`
`HAVE_PRE_DECREMENT`

Similar for other kinds of addressing.

`CONSTANT_ADDRESS_P (`*x*`)`

A C expression that is 1 if the RTX *x* is a constant whose value is an integer. This includes integers whose values are not explicitly known, such as '`symbol_ref`' and '`label_ref`' expressions and '`const`' arithmetic expressions.

`MAX_REGS_PER_ADDRESS`

A number, the maximum number of registers that can appear in a valid memory address.

`GO_IF_LEGITIMATE_ADDRESS (`*mode*`, `*x*`, `*label*`)`

A C compound statement with a conditional `goto` *label*; executed if *x* (an RTX) is a legitimate memory address on the target machine for a memory operand of mode *mode*.

It usually pays to define several simpler macros to serve as subroutines for this one. Otherwise it may be too complicated to understand.

`LEGITIMIZE_ADDRESS (`*x*`, `*oldx*`, `*mode*`, `*win*`)`

A C compound statement that attempts to replace *x* with a valid memory address for an operand of mode *mode*. *win* will be a C statement label elsewhere in the code; the macro definition may use

```
GO_IF_LEGITIMATE_ADDRESS (mode, x, win);
```

to avoid further processing if the address has become legitimate.

x will always be the result of a call to `break_out_memory_refs`, and *oldx* will be the operand that was given to that function to produce *x*.

The code generated by this macro should not alter the substructure of *x*. If it transforms *x* into a more legitimate form, it should assign *x* (which will always be a C variable) a new value.

It is not necessary for this macro to come up with a legitimate address. The compiler has standard ways of doing so in all cases. In fact, it is safe for this macro to do nothing. But often a machine-dependent strategy can generate better code.

9.7 Miscellaneous Parameters

`CASE_VECTOR_MODE`

An alias for a machine mode name. This is the machine mode that elements of a jump-table should have.

`CASE_VECTOR_PC_RELATIVE`

Define this macro if jump-tables should contain relative addresses.

`IMPLICIT_FIX_EXPR`

An alias for a tree code that should be used by default for conversion of floating point values to fixed point. Normally, `FIX_ROUND_EXPR` is used.

`EASY_DIV_EXPR`

An alias for a tree code that is the easiest kind of division to compile code for in the general case. It may be `TRUNC_DIV_EXPR`, `FLOOR_DIV_EXPR`, `CEIL_DIV_EXPR` or `ROUND_DIV_EXPR`. These four division operators differ in how they round the result to an integer. `EASY_DIV_EXPR` is used when it is permissible to use any of those kinds of division and the choice should be made on the basis of efficiency.

`MOVE_MAX` The maximum number of bytes that a single instruction can move quickly from memory to memory.

`SLOW_ZERO_EXTEND`

Define this macro if zero-extension (of chars or shorts to integers) can be done faster if the destination is a register that is known to be zero.

`SHIFT_COUNT_TRUNCATED`

Define this macro if shift instructions ignore all but the lowest few bits of the shift count. It implies that a sign-extend or zero-extend instruction for the shift count can be omitted.

`TRULY_NOOP_TRUNCATION` (*outprec*, *inprec*)

A C expression which is nonzero if on this machine it is safe to "convert" an integer of *inprec* bits to one of *outprec* bits (where *outprec* is smaller than *inprec*) by merely operating on it as if it had only *inprec* bits.

On many machines, this expression can be 1.

`STORE_FLAG_VALUE`

A C expression for the value stored by a store-flag instruction (*scond*) when the condition is true. This is usually 1 or -1; it is required to be an odd number.

Do not define `STORE_FLAG_VALUE` if the machine has no store-flag instructions.

`Pmode` An alias for the machine mode for pointers. Normally the definition can be

```
#define Pmode SImode
```

`FUNCTION_MODE`

An alias for the machine mode used for memory references to functions being called, in '`call`' RTL expressions. On most machines this should be `QImode`.

`CONST_COST` (*x*, *code*)

A part of a C `switch` statement that describes the relative costs of constant RTL

expressions. It must contain `case` labels for expression codes '`const_int`', '`const`', '`symbol_ref`', '`label_ref`' and '`const_double`'. Each case must ultimately reach a `return` statement to return the relative cost of the use of that kind of constant value in an expression. The cost may depend on the precise value of the constant, which is available for examination in *x*.

code is the expression code—redundant, since it can be obtained with `GET_CODE (`*x*`)`.

9.8 Condition Code Information

The file '`conditions.h`' defines a variable `cc_status` to describe how the condition code was computed (in case the interpretation of the condition code depends on the instruction that it was set by). This variable contains the RTL expressions on which the condition code is currently based, and several standard flags.

Sometimes additional machine-specific flags must be defined in the machine description header file. It can also add additional machine-specific information by defining `CC_STATUS_MDEP`.

`CC_STATUS_MDEP`

C code for a data type. This type is used for declaring the `mdep` component of `cc_status`. It defaults to `int`.

`CC_STATUS_MDEP_INIT`

A C expression for the initial value of the `mdep` field. It defaults to 0.

`NOTICE_UPDATE_CC (`*exp*`)`

A C compound statement to set the components of `cc_status` appropriately for an insn whose body is *exp*. It is this macro's responsibility to recognize insns that set the condition code as a byproduct of other activity as well as those that explicitly set `(cc0)`.

If there are insn that do not set the condition code but do alter other machine registers, this macro must check to see whether they invalidate the expressions that the condition code is recorded as reflecting. For example, on the 68000, insns that store in address registers do not set the condition code, which means that usually `NOTICE_UPDATE_CC` can leave `cc_status` unaltered for such insns. But suppose that the previous insn set the condition code based on location '`a4@(102)`' and the current insn stores a new value in '`a4`'. Although the condition code is not changed by this, it will no longer be true that it reflects the contents of '`a4@(102)`'. Therefore, `NOTICE_UPDATE_CC` must alter `cc_status` in this case to say that nothing is known about the condition code value.

9.9 Output of Assembler Code

`ASM_SPEC` A C string constant that tells the GNU CC driver program switches to pass to the assembler. It can also specify how to translate switches you give to GNU CC into switches for GNU CC to pass to the assembler. See the file '`tm-sun3.h`' for an example of this.

Do not define this macro if it does not need to do anything.

`LINK_SPEC`

A C string constant that tells the GNU CC driver program switches to pass to the linker. It can also specify how to translate switches you give to GNU CC into switches for GNU CC to pass to the linker.

Do not define this macro if it does not need to do anything.

`ASM_FILE_START`

A C string constant for text to be output at the start of each assembler output file. Normally this is `"#NO_APP"`, which is a comment that has no effect on most assemblers but tells the GNU assembler that it can save time by not checking for certain assembler constructs.

`ASM_APP_ON`

A C string constant for text to be output before each `asm` statement or group of consecutive ones. Normally this is `"#APP"`, which is a comment that has no effect on most assemblers but tells the GNU assembler that it must check the lines that follow for all valid assembler constructs.

`ASM_APP_OFF`

A C string constant for text to be output after each `asm` statement or group of consecutive ones. Normally this is `"#NO_APP"`, which tells the GNU assembler to resume making the time-saving assumptions that are valid for ordinary compiler output.

`TEXT_SECTION_ASM_OP`

A C string constant for the assembler operation that should precede instructions and read-only data. Normally `".text"` is right.

`DATA_SECTION_ASM_OP`

A C string constant for the assembler operation to identify the following data as writable initialized data. Normally `".data"` is right.

`REGISTER_NAMES`

A C initializer containing the assembler's names for the machine registers, each one as a C string constant. This is what translates register numbers in the compiler into assembler language.

`DBX_REGISTER_NUMBER (`*regno*`)`

A C expression that returns the DBX register number for the compiler register number *regno*. In simple cases, the value of this expression may be *regno* itself. But sometimes

there are some registers that the compiler knows about and DBX does not, or vice versa. In such cases, some register may need to have one number in the compiler and another for DBX.

`ASM_OUTPUT_DOUBLE` (*file*, *value*)

A C statement to output to the stdio stream *file* an assembler instruction to assemble a `double` constant whose value is *value*. *value* will be a C expression of type `double`.

`ASM_OUTPUT_FLOAT` (*file*, *value*)

A C statement to output to the stdio stream *file* an assembler instruction to assemble a `float` constant whose value is *value*. *value* will be a C expression of type `float`.

`ASM_OUTPUT_ASCII` (*file*, *ptr*, *len*)

A C statement to output to the stdio stream *file* an assembler instruction to assemble a string constant containing the *len* bytes at *ptr*. *ptr* will be a C expression of type `char *` and *len* a C expression of type `int`.

If the assembler has a `.ascii` pseudo-op as found in the Berkeley Unix assembler, do not define the macro `ASM_OUTPUT_ASCII`.

`ASM_OUTPUT_SKIP` (*file*, *nbytes*)

A C statement to output to the stdio stream *file* an assembler instruction to advance the location counter by *nbytes* bytes. *nbytes* will be a C expression of type `int`.

`ASM_OUTPUT_ALIGN` (*file*, *power*)

A C statement to output to the stdio stream *file* an assembler instruction to advance the location counter to a multiple of 2 to the *power* bytes. *power* will be a C expression of type `int`.

`ASM_INT_OP`

A C string constant for the assembler operation that assembles constants of C type `int`. A space must follow the operation name. Normally `".long "`.

`ASM_SHORT_OP`

`ASM_CHAR_OP`

Likewise, for C types `short` and `char`. Normally `".word "` and `".byte "`.

`TARGET_BELL`

A C constant expression for the integer value for escape sequence '`\a`'.

`TARGET_BS`

`TARGET_TAB`

`TARGET_NEWLINE`

C constant expressions for the integer values for escape sequences '`\b`', '`\t`' and '`\n`'.

`TARGET_VT`

`TARGET_FF`

`TARGET_CR`

C constant expressions for the integer values for escape sequences '`\v`', '`\f`' and '`\r`'.

`PRINT_OPERAND` (*file*, *x*, *code*)

A C compound statement to output to stdio stream *file* the assembler syntax for an instruction operand *x*. *x* is an RTL expression.

code is a value that can be used to specify one of several ways of printing the operand. It is used when identical operands must be printed differently depending on the context. *code* comes from the '%' specification that was used to request printing of the operand. If the specification was just '%*digit*' then *code* is 0; if the specification was '%*ltr digit*' then *code* is the ASCII code for *ltr*.

If *x* is a register, this macro should print the register's name. The names can be found in an array `reg_names` whose type is `char *[]`. `reg_names` is initialized from `REGISTER_NAMES`.

`PRINT_OPERAND_ADDRESS` (*file*, *x*)

A C compound statement to output to stdio stream *file* the assembler syntax for an instruction operand that is a memory reference whose address is *x*. *x* is an RTL expression.